LEGO® MindStorms Scout

Software Developers Kit

User Guide & Reference

Foreword

At LEGO, we believe that imagination is important to the world. For decades, the LEGO construction materials have been a means for people of all ages to express creativity and make discoveries of their own. The addition of LEGO programmable bricks brings a whole new dimension to construction.

The LEGO programmable bricks are microcomputers, which makes it possible to add functions or behavior to physical creations made by LEGO pieces. The functions or behavior are controlled by means of programming.

LEGO has launched a new programmable brick: the Scout™ of LEGO® MINDSTORMS™ Robotics Discovery System™. The programming software codes of this product have deliberately been designed to be easy to use - yet versatile and powerful in function. This has been important to enable kids to use the new technology for creation of their own personally meaningful inventions.

This technical reference guide is published to allow more creative freedom in the programming for more experienced users. The technical reference guide documents how the Scout™ can be programmed by means of LEGO Assembly programs. We hope that the release of this document will inspire even more people to develop imaginative applications of the Scout™.

We kindly ask you to read the License Agreement and Warranty Disclaimer below before using this document.

We wish you good luck with development of creative applications.

LEGO - just imagine...

SOFTWARE DEVELOPER KIT LICENSE AGREEMENT AND WARRANTY DISCLAIMER

License for the Software included in the LEGO MINDSTORMS Software Developer Kit (hereinafter referred to as the Software) from the LEGO Group.

IMPORTANT -- READ CAREFULLY: By using the information contained in this document you agree to be and are hereby bound by the terms of this License Agreement. If you do not agree to the terms of this Agreement, do not use the information contained in this document.

I. GRANT OF LICENSE:

The LEGO Group and its suppliers and licensors (hereinafter referred to as LEGO) hereby grant you a non-exclusive, non-commercial license to use the Software subject to the following terms:

You may:
- (i) use the Software only to develop applications for the LEGO MINDSTORMS Scout;
- (ii) the applications developed by means of the Software or parts hereof shall only be used for purposes that neither directly nor indirectly have any commercial implications;

You may not:

- (i) permit other individuals to use the Software except under the terms listed above;
- (ii) modify, translate, reverse engineer, decompile, disassemble (except to the extent that this restriction is expressly prohibited by law) or create derivative works based upon the Software;
- (iii) resell, rent, lease, transfer, or otherwise transfer rights to the Software; or
- (v) remove any proprietary notices or labels on the Software.

II. ENHANCEMENTS OR UP-DATES:

This license does not grant you any right to any enhancement or up-date.

III. TITLE:

Title, ownership, rights, and intellectual property rights in and to the Software shall remain with the LEGO Group. The Software is protected by national copyright laws and international copyright treaties. The communication protocol is protected by a pending patent application.

Title, ownership rights and intellectual property rights in and to the content accessed through the Software including any content contained in the Software media demonstration files is the property of the applicable content owner and may be protected by applicable copyright or other law. This license gives you no rights to such content.

LEGO, the LEGO logo, the LEGO Brick and LEGO MINDSTORMS are some of the trademarks belonging exclusively to the LEGO Group.

If you want to learn more about how to use trademarks and other proprietary rights belonging to the LEGO Group please visit our web site: http://www.lego.com.

All other trademarks mentioned in this document are the property of their respective owners.

IV. DISCLAIMER OF WARRANTY:

THE SOFTWARE IS PROVIDED FOR FREE WITHOUT ANY KIND OF MAINTAINANCE OR SUPPORT.

THE SOFTWARE IS PROVIDED AS IS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, THE LEGO GROUP FURTHER DISCLAIMS ALL WARRANTIES, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT. THE ENTIRE RISK ARISING OUT OF THE USE OR PERFORMANCE OF THE SOFTWARE OR APPLICATIONS DEVELOPED BY MEANS OF THE SOFTWARE REMAINS WITH YOU. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL THE LEGO GROUP OR ITS SUPPLIERS BE LIABLE FOR ANY CONSEQUENTIAL, INCIDENTAL, DIRECT, INDIRECT, SPECIAL, PUNITIVE, OR OTHER DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR OTHER PECUNIARY LOSS) ARISING OUT OF THIS AGREEMENT OR THE USE OF OR INABILITY TO USE THE PRODUCT, EVEN IF THE LEGO GROUP HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME STATES/JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

V. TERMINATION:

This license shall terminate automatically if you fail to comply with the limitations described in this Agreement. No notice shall be required from the LEGO Group to effectuate such termination. On termination you must destroy all copies of the Software and applications developed by means of the Software.

VI. GOVERNING LAW:

This License Agreement shall be governed by the laws of the jurisdiction, where you have permanent residency. However, if the product is bought in USA the License Agreement shall be governed by the laws of the State of Connecticut, without regard to conflicts of law provisions, and if the product in bought in USA you consent to the exclusive jurisdiction of the state and federal courts sitting in the State of Connecticut. This License Agreement will not be governed by the United Nations Convention of Contracts for the International Sale of Goods, the application of which is hereby expressly excluded.

VII. ENTIRE AGREEMENT:

This Agreement constitutes the complete and exclusive agreement between the LEGO Group and you with respect to the subject matter hereof and supersedes all prior oral or written understandings, communications or agreements not specifically incorporated herein. This Agreement may not be modified except in writing duly signed by an authorized representative of the LEGO Group and you.

Table of Contents

Introduction

This User Guide & Reference document tells you how to use ScoutTool.exe and ScoutDOS.exe applications directly to write programs for the LEGO® MINDSTORMS™ Scout™ programmable brick.

All examples in this document are written as LEGO Assembly (LASM) programs, which is a text representation of the byte code commands that the Scout can execute, providing detailed control over the Scout.

Pre-requisites

No other software is required but you must have a LEGO serial cable and an IR Tower from another MindStorms set such as LEGO MindStorms Robotics Invention System 1.0 or 1.5. You may also purchase a combined IR Tower and cable pack (item W979713) over the Internet from www.pitsco-legodacta-store.com (select “RoboLab”, then “RoboLab Components” and finally “Infrared Transmitter and Cable Pack (PC and MAC)”).

Document structure

The rest of the document contains:

- a more technical description of the Scout,
- how to install and use the SDK,
- what byte code assembly commands are available and what they do,
- standard program structure templates, and
- general programming issues to consider and be aware of when controlling robot systems.

This document is not intended to be a complete textbook on the art of programming or the art of robotics or any such related area. Instead it is hoped that it will provide a correct technical description of how to program the Scout at the most detailed level in order to get as much as possible out of all the functionality it provides.

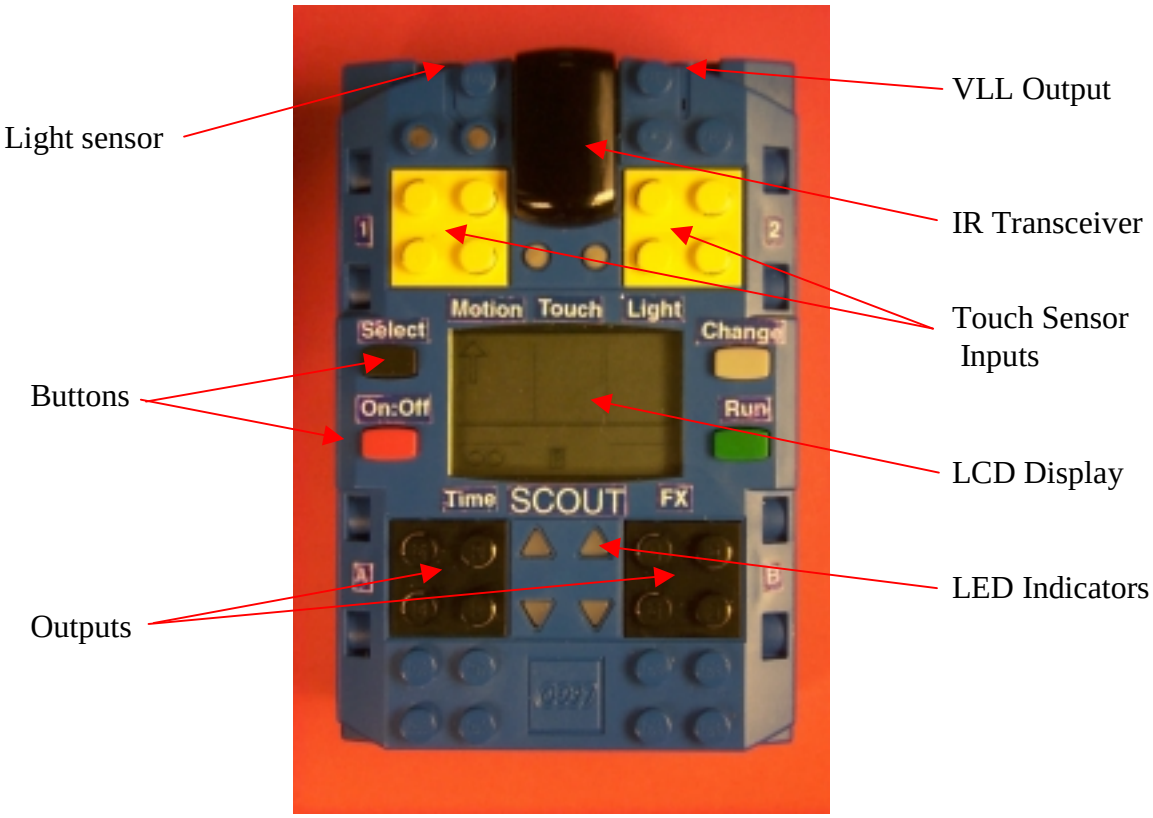
The Scout – brief description

In this part a brief introduction is given to the basic concepts of the Scout.

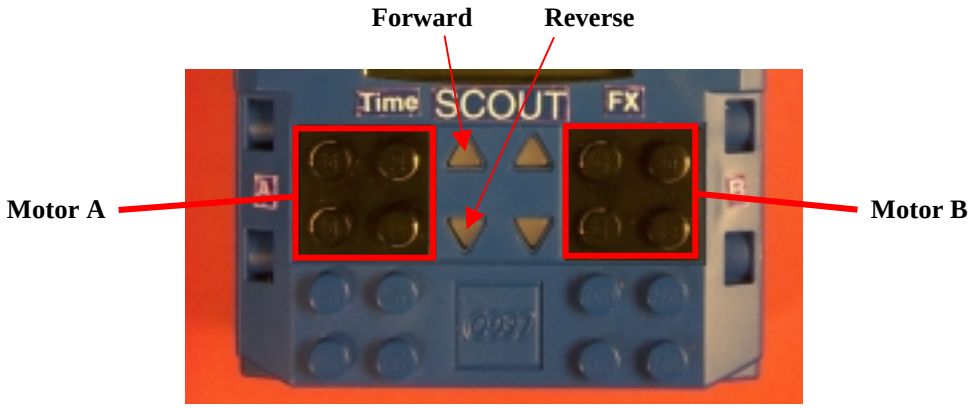
The Scout is developed with the RCX as a 'big brother' but targeted at lower age and lower price. Therefore a lot of the elements of the RCX are reused, some are left out and a range of new elements is put in, partly to accommodate the lower age, partly from hard earned experiences with the RCX.

The Scout brick seen from the outside

The size of the Scout is the same as the RCX. It uses the same battery box, containing 6 AA battery cells. Also the placement of buttons, sensor inputs and motor outputs is consistent with the design of the RCX.

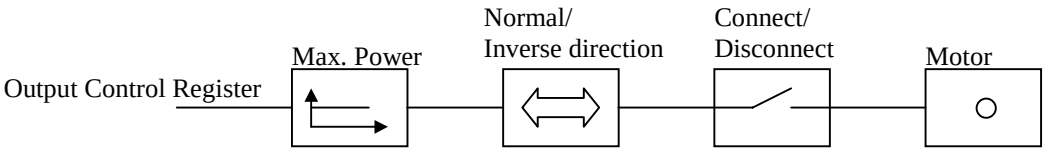


Output ports with LED indicators



The Scout has two motor output ports: Motor A and Motor B. Connection to the ports is made with the standard LEGO 2-wire terminal system. Motors can be turned On and Off and their direction and power level can be set.

As a global control you can connect and disconnect a motor, set normal or inverse direction and set a maximum power level. Global motor control works as illustrated bellow:



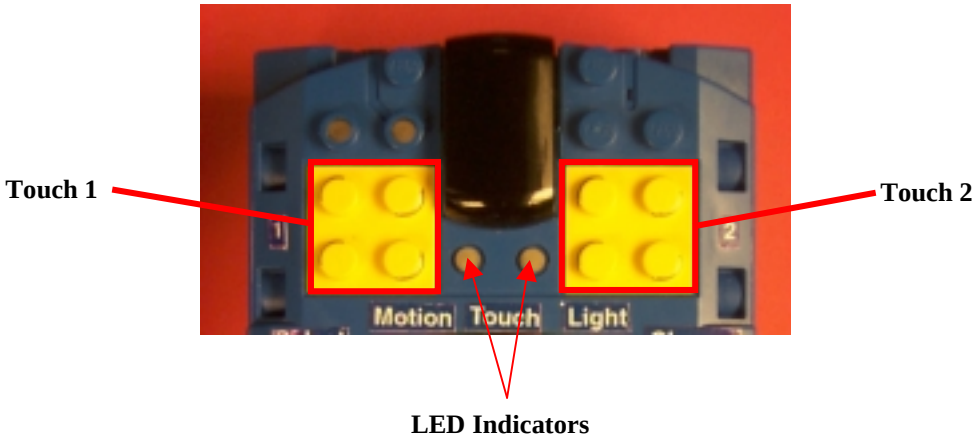
Power to the motor output ports is selectable in 8 levels. In Stand Alone mode (see Stand Alone Mode) the motors always run at full power. Direction of the motors can be set to Forward or Reverse.

Control of the motor outputs can be obtained in four different ways:

- From a downloaded program
- By direct command from the IR Tower
- Using the LEGO Remote Control
- By PB Message from another P-Brick

To each motor output there are two green LED's indicating motor direction when the output is ON. Try to connect a motor to an output. Don't turn the Scout On. Then turn the motor and look at the indicator LED's. Isn't that cool?

Touch sensor ports with LED indicators



The Scout has two touch sensor input ports: Touch 1 and Touch 2. Connection to the ports is made with the standard LEGO 2-wire terminal system. Only the LEGO touch sensor type, with optional resistor identification, is supported at these inputs.

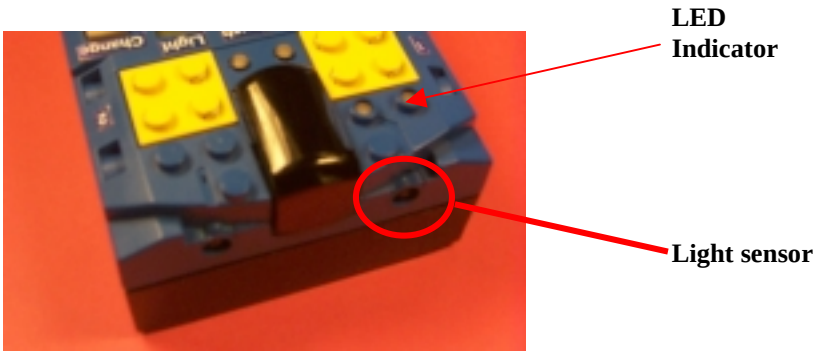
Sensor values are:

Boolean value:	0 or 1	(Released or Pressed)
Raw value:	0 to 255	

The Sensor ID can be read in the Sensor Type Register. In this way the program inside the Scout or an application is able to see which kind of touch sensor is connected to a certain input. Three ID-touch sensors are available: Red, Yellow or White.

For each touch sensor a yellow LED is indicating when the sensor is activated (pressed).

Build-in light input with LED indicator



The light sensor performs measurement of the surrounding light level.

Values of the sensor are:

State value:	Dark, Normal or Bright These states are determined using an upper threshold, a lower threshold and a hysteresis set up for the light sensor: When the raw sensor value is below lower threshold the state is Bright, above the upper threshold it is Dark, else it is Normal.
Raw value:	0-1020 (Low value: Bright, high value: Dark)

In Stand Alone mode the light sensor is automatically calibrated at start up. The surrounding light level is measured and upper- and lower threshold is set 12.5% above and below this center value. Hysteresis is set to 3.12% of the center value.

A yellow LED indicates when the light sensor is in Dark or Bright states.

VLL output

The Visible Light Link (VLL) output can send VLL codes.

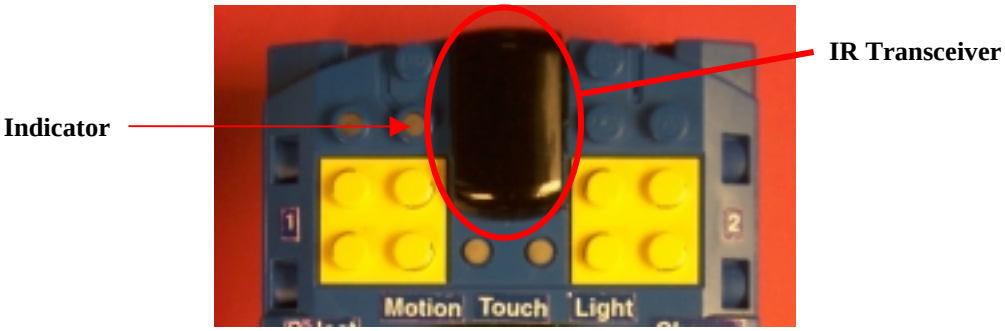
The VLL output can be controlled from the user program or through direct IR commands. If a Motor C button on the LEGO Remote Control is pressed, the corresponding motor control command will be sent on the VLL output.

The VLL signals are transmitted using visible red light through an optical fiber.

The MicroScout has VLL-input. Motor and sound can be controlled and a program can be scripted into the MicroScout through the VLL link. This can be done from a program in the Scout, or by using the Scout as the "intermediate agent" it can be done from an application, from the LEGO remote control or from another P-Brick (Scout/RCX).

In the future more bricks equipped with VLL input/output may appear.

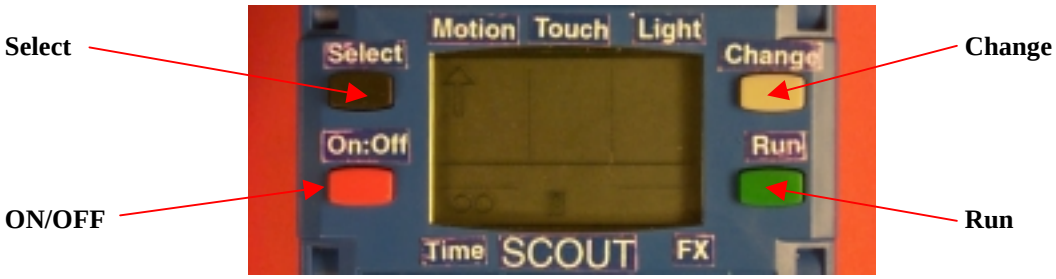
IR transceiver with LED indicator



The IR transceiver unit is used to communicate with a PC through the IR Tower (using the Scout low-level protocol), to receive remote control commands and to communicate with other P-Bricks through the PB Message system.

A green LED indicates IR Transceiving.

Buttons: ON/OFF, Select, Change, Run



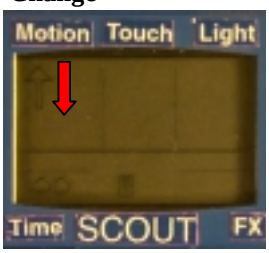
Four buttons and an LCD display form the primary user interface.

Buttons are:

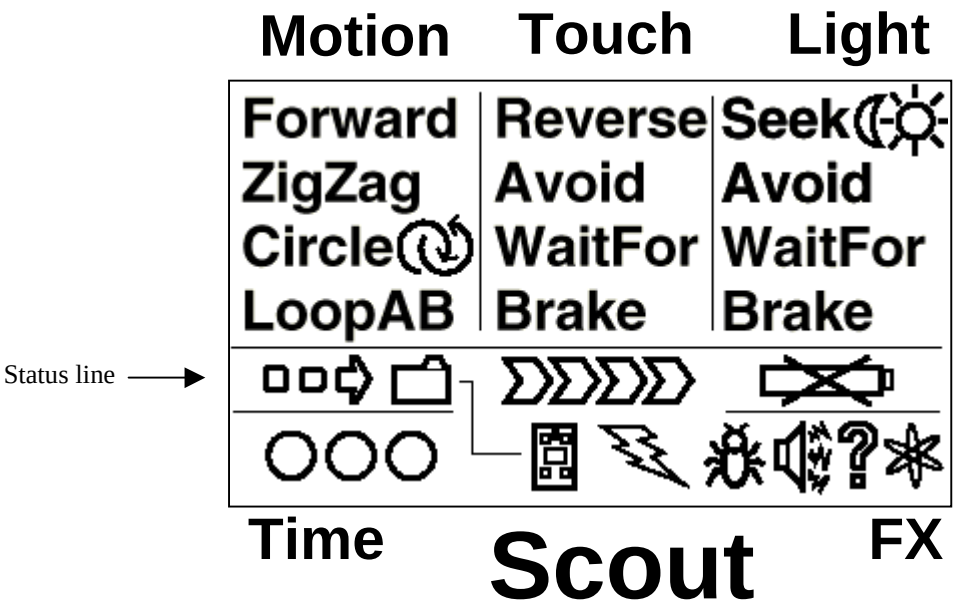
- **On-Off**
  Turns the Scout ON or OFF. Press the On/Off-button and hold it down a couple of seconds to **reset** the Scout.
- **Select**
  Moves the focus to the next group on the display
- **Change**
  Changes the setting of the group in focus
- **Run**
  Runs the program, if any, of the selected mode

Select

Change



LCD display

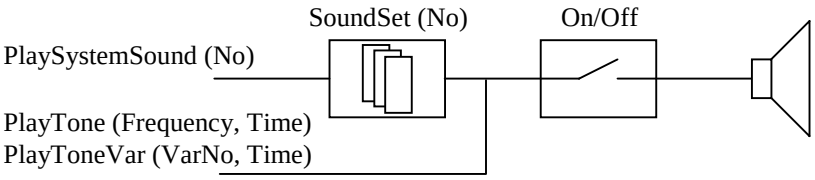


The LCD display is divided into seven groups:

- **Motion**
  Selection of basic motion patterns in Stand Alone mode
- **Touch**
  Selection of touch sensor rules in Stand Alone mode
- **Light**
  Selection of light sensor rules in Stand Alone mode
- **Time**
  Selection of time scaling factors in Stand Alone mode
- **FX**
  Selection of special effects theme in Stand Alone mode
- **Scout**
  Mode selection: Stand Alone – or Power mode
- **Status**
  Display of status information:
  Download indicator, Power mode program folder, Run indicator, Battery low symbol

Sound output

To output sound the Scout is equipped with a piezo sound element.
Sounds can be played by using the built in system sounds with the sound sets or by playing a tone with a certain frequency (as a constant or from a variable) for a certain time:



Sound can globally be turned On and Off.

Inside the Scout: Basic functionality

The Scout is operated in two different modes: Stand Alone mode and Power mode.
The user sets the mode of operation by selecting the Scout-group on the display and then change to the desired mode.

Stand Alone Mode

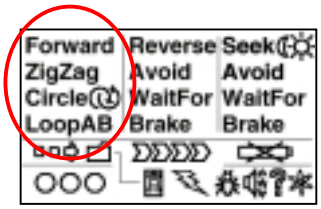
The first thing you will notice turning the Scout On in Stand Alone mode is the sound of the heartbeat. A subtle beating sound in the background telling you, that the Scout is alive.

In this mode the Scout has full IR-link capabilities for direct control, but primary user interaction and programming is performed through buttons and LCD-display. No programs can be downloaded.

Choosing one and only one rule (command) from each programming group does programming of the Scout. You use the Select and Change buttons to navigate the groups of the display and change the selection in the groups. When the rule selection is done, press the Run button and the Stand Alone program will be running.

In the **Motion** group the basic default motion pattern is selected. The options are:

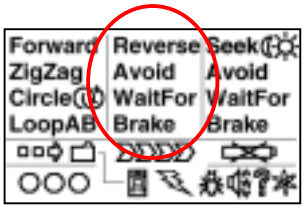
- No Motion (0)
- Forward (1)
- ZigZag (2)
- Circle Right (3)
- Circle Left (4)
- Loop A (5)
- Loop B (6)
- Loop AB (7)



When an event triggers a rule from one of the other groups, and motor control is wanted, the default motion is interrupted. After this action is completed the Scout will return to default motion. When No Motion is selected the Scout will simply be waiting for the event driven action of the other groups to happen.

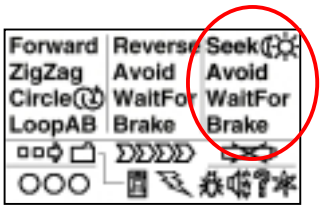
In the **Touch** group the rule for the two touch sensors is selected. All the rules uses both touch sensors. The rules are:

- Ignore (0)
- Reverse (1)
  When T1 or T2 is hit direction on both motors is changed
- Avoid (2)
  When T1 is hit the model will back up and turn to the right
  When T2 is hit the model will back up and turn to the left
- WaitFor (3)
  The model waits for T1 or T2 to get hit, then action starts
- Brake (4)
  While T1 is pressed, Motor A is braked
  While T2 is pressed, Motor B is braked



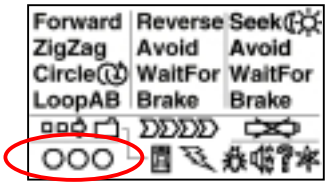
The **Light** group contains the light sensor rules:

- Ignore (0)
- Seek Light (1)
  Model turns around every now and then and finds the brightest direction
- Seek Dark (2)
  Model turns around every now and then and finds the darkest direction
- Avoid (3)
  If it gets Bright or Dark the model will back up and turn away
- WaitFor (4)
  The model waits for the light to get Bright or Dark, then action starts
- Brake (5)
  While the light is Bright or Dark both motors are braked



In the **Time** group you select the Time Scaling factor of the Stand Alone Program System. In all of the Program Blocks used in the Stand Alone Program System timing values are scaled with this factor. Settings are:

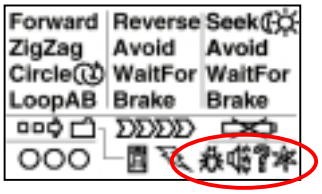
- Short (0, one circle) Scale factor 1
- Medium (1, two circles) Scale factor 2
- Long (2, three circles) Scale factor 4



I.e. the fewer circles, the faster the program will run.

In the **FX** group a Special Effects Theme is selected. Each theme has a Sound Set that will take effect on all the event triggered sounds in the Scout. FX Theme selections:

- No Theme (0)
- Bug (1) Does the Bug-dance every now and then
- Alarm (2) Alarm-sound depending on the motor drive
- Random (3) Does a sequence of random movements now and then
- Science (4) Sound beeping Geiger function on the light sensor



When the program runs in Stand Alone mode, a background task is waiting for PB Messages and does actions depending on the incoming message value. The messages 1, 2 and 3 are used to start a special effect:

PBM 1:	Seek dark
PBM 2:	Seek light
PBM 3:	The Bug dance

When the special effect is finished the Scout will return to the default motion (as with any FX selected in the program).

Messages 4-12 are used to do basic motor drive control:

PBM 4:	Forward	A Fwd, B Fwd
PBM 5:	Reverse	A Rwd, B Rwd
PBM 6:	Spin right	A Fwd, B Rwd
PBM 7:	Spin left	A Rwd, B Fwd
PBM 8:	Turn right (forward)	A Fwd, B Off
PBM 9:	Turn left (reverse)	A Rwd, B Off
PBM 10:	Turn left (forward)	A Off, B Fwd
PBM 11:	Turn right (reverse)	A Off, B Rwd
PBM 12:	Stop	A Off, B Off

After half a second the motor drive started by a PB Message will time out and the Scout will return to the default motion. In this way an RCX can repeatedly send PB Messages to stay in control. The motor control made by the RCX will override the

default motion. If a sensor event or a special effect takes place it will interrupt the PB Message controlled motion (just as it would with the default motion).

You could e.g. program a Scout for Forward motion and Avoid touch. Then program an RCX to send PBM 10 repeatedly when Touch sensor 1 is pressed and PBM 8 when Touch sensor 3 is pressed. Thus the RCX can be used to remote steer the Scout by overriding the Scout default motion. If one of the touch sensors on the Scout is hit, the Avoid sequence will be started, even if the remote steering from the RCX is active.

Power Mode

In Power mode the user can interact with the Scout from an application through e.g. the LEGO IR Tower. The communication is done using the IR link and the Scout Low Level protocol.

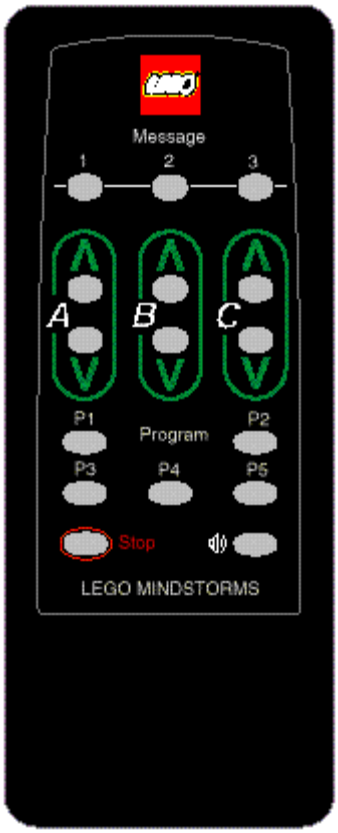
In this mode you can perform direct control of the Scout from an application, you can poll data from the Scout and you can download a program to be run from the Power mode Program System.

The VLL Output can be used for programming or controlling other devices equipped with VLL Input.

Pressing the Run button in Power mode runs the Power mode program if any.
Pressing the Select button will have no effect (Error sound).
Pressing the Change button will change to the Stand Alone mode.

The Scout and the LEGO Remote

The LEGO Remote Control can be used to control the Scout without having to touch the buttons. PB Messages can be sent, basic motor control can be performed, programs can be started and stopped and a sound can be played.



Button	Action
Message 1	PB Message 1 is sent: SeekDark
Message 2	PB Message 2 is sent: SeekLight
Message 3	PB Message 3 is sent: BugDance
A ▲	Motor A Forward while button is pressed
A ▼	Motor A Backward while button is pressed
B ▲	Motor B Forward while button is pressed
B ▼	Motor B Backward while button is pressed
C ▲	VLL output (red LED/Motor C) sends Forward command while button is pressed
C ▼	VLL output (red LED/Motor C) sends Backwards command while button is pressed
Sound	Play RemoteSound
Stop	Stop running program (Stop all tasks)

The program buttons are used to run programs:

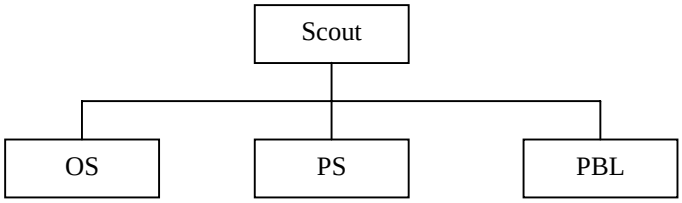
P1-P4 Set Scout in Stand Alone mode
Stand Alone Setup according to table bellow
Run Stand Alone program

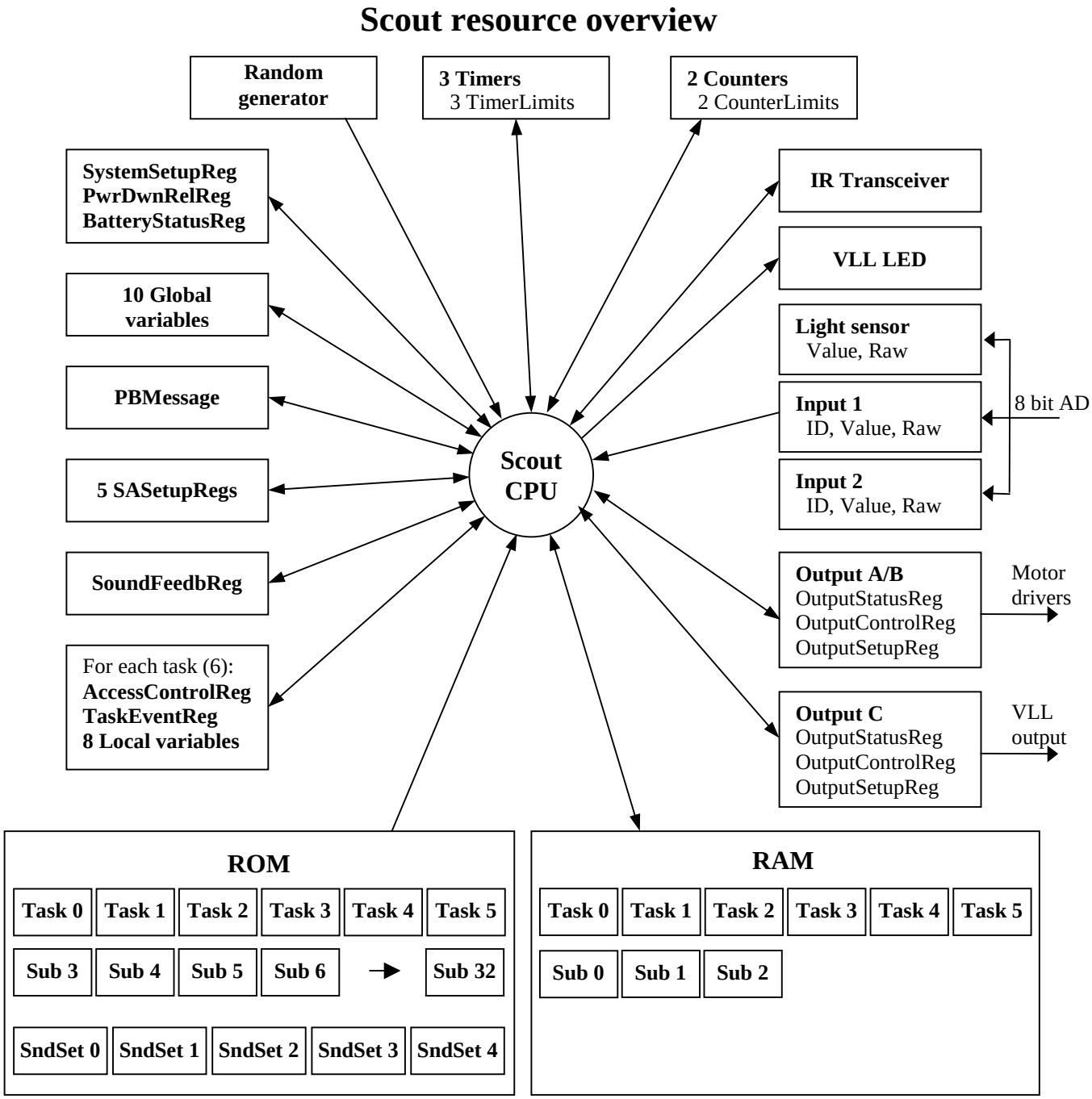
Program	Model	Motion	Touch	Light	Time	FX
P1	Bug 1	ZigZag	Avoid	Ignore	Short	Bug
P2	Bug 2	Forward	Avoid	SeekLight	Medium	Bug
P3	Intruder Alarm	Loop AB	Ignore	WaitFor	Medium	Alarm
P4	Hoop-o-bot	Loop AB	Ignore	Ignore	Short	Random

P5 Set Scout in Power mode
If Power mode program is present (Task 0 is not empty):
Run Power mode program (Start Task 0)

Firmware system design - overview

In this part an overview is given on the structure of the firmware system. The Scout at top level consists of three parts: The **Operating System**, the **Program System** and a **Program Block Library** (Subs):





The Operating System

The **Operating System** performs:

- Hardware I/O
  Sensor input, Motor output, Sound output, VLL output, IR I/O, Button input, LCD display output.
- Basic system control
  Scheduling execution, battery surveillance, power down control etc.
- Control of Downloading in Power mode
- Interpretation of Scout Low Level Commands
  From direct control or from program control.
- Execution of the **Program System**
  Running programs in Stand Alone –, or Power mode.
- System control through registers
- **Priority** control
- **Event** generation, sound feedback.

The Program System

The **Program System** has two different modes of operation reflecting the Stand Alone – and Power modes of the Scout.

- **Stand Alone** mode:
  The PS is run from a set of ROM Tasks and is "programmed" by setting up the 5 Stand Alone Setup values: Motion, Touch, Light, Time and FX. A subset of the Program Blocks in the library is used in Stand Alone mode.

- **Power** mode:
  The PS runs from a set of downloaded RAM Tasks. All of the Subs in the Program Block Library can be freely used. Also user Subs can be downloaded and used from the RAM Tasks.

The Program System interfaces to the Operating System through a number of global and mode specific resources.

System overview

Tasks

A program consists of one or more tasks running in parallel. Pressing the Run button starts Task 0.

A task contains a stack of commands that are executed in a sequence. A task can start and stop other tasks and call subroutines.

- The Stand Alone program consists of 6 fixed tasks placed in ROM. These tasks cannot be accessed in Power mode.
- A Power mode program consists of up to 6 tasks placed in RAM.

Subroutines

A subroutine contains a stack of commands that are executed in a sequence.

When execution of the subroutine reaches the end, program execution is returned to the task that called the subroutine (at the point just after the call). A subroutine cannot call another subroutine.

- The Scout has a library of fixed subroutines in ROM.
- Up to three subroutines can be downloaded to RAM.

Global set up

Global settings of the Scout:

- Select the mode of the Scout (From PC or via button interface)
- Turn the sound On and Off (From PC or within program)
- Select one of 5 sound sets (From PC or within program)
- Set the power down time (From PC or within program)
- Select short or long IR range (Only from PC)

Resources available to the user

Hardware resources

Hardware resources are the physical devices available from the outside of the Scout.

Inputs:

Touch sensor 1 and 2

Values are:

- Touch sensor state value: Pressed or Released
- ID number of the touch sensor connected to the port: ID0, ID1 or ID2

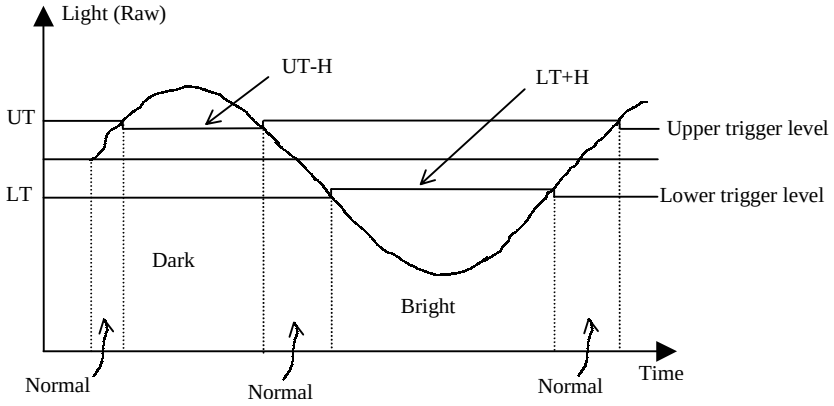
Light sensor

Values are:

- Light sensor state value: Dark, Normal, Bright or Undefined
- Light sensor raw value: 0-1020 (Low value: Bright, high value: Dark)

The state values of the light sensor are generated from the raw value passing upper and lower trigger levels. These trigger levels are calculated using three light sensor parameters:

- Upper Threshold UT
- Lower Threshold LT
- Hysteresis H



Outputs:

Motor A, B and C

Motor C is directed to the VLL output.

Immediate motor state settings:

On/Off
Forward/Reverse direction
Power level 1-8 (not motor C)

Global motor settings:

Connect/disconnect
Normal/Inverse direction
Maximum power level 1-8 (not motor C)

The current setting of a motor can be read in the Motor Status Register.

Sound

Sound is controlled in three ways:

Play one of 28 system sounds
System sounds 0 to 9 are placed outside the sound set system.
System sounds 10 to 27 are sound set dependable.
5 fixed sound sets are available.

Play a tone for some time
Give a fixed tone frequency or take the frequency from a variable.

Let the operating system give sound feedback on events.
System sounds 10 to 24 are used by the operating system to give sound feedback on the 15 Scout events.
System sound 25 is the special 'Dance' sound. System sound 26 is the special 'Bug' sound. System sound 27 is the special 'Random' sound.
Timer 0 event is used to generate the heartbeat.

Feedback on each event can be turned On and Off.

Visible light link (VLL)

Turn the VLL diode On or Off
Send a VLL code.

Software resources

Software resources are those available inside the Scout system.

Variables

- You can load a variable with a constant or copy almost any Scout system value.
- You can perform mathematical operations on and between variables (add, subtract, multiply, divide, AND, OR, etc.).

Global variables:

- 10 global variables (all tasks can see them)

Local variables:

- 8 local variables (each task can only see its own locals)

Stand Alone programming variables

- 5 stand alone programming variables make up the programming of the Stand Alone Scout.

Can be set through the button interface in Stand Alone mode or by command in Power mode.

Random generator

- Can generate a random number between 0 and X.

Timers

- 3 timers each with a user defined timer limit.
- Resolution of the timers is 100ms.
- A timer runs to its limit and is then reset to zero by the operating system.
- When a timer reaches its limit, it generates a Timer X on limit event.
- You can copy a timer value into a variable and you can reset a timer to zero.

Counters

- 2 counters each with a user defined counter limit.
- When a counter reaches its limit, it generates a Counter X on limit event.
- You can copy a counter value into a variable, you can increment or decrement it by one or you can reset it to zero.

Mailbox

- One mailbox, that can be loaded (via IR) with a message value between 1 and 255.
- You can copy the contents of the mailbox into a variable or you can reset the contents to zero.

Scout events

- 15 events

    1. Touch 1 pressed
    2. Touch 1 released
    3. Touch 2 pressed
    4. Touch 2 released
    5. Light entering Light state
    6. Light entering Normal state
    7. Light entering Dark state
    8. 1 Blink detected (a blink time can be set up)
    9. 2 Blinks detected
    10. Counter 0 on limit
    11. Counter 1 on limit
    12. Timer 0 on limit
    13. Timer 1 on limit
    14. Timer 2 on limit
    15. Mail received
- Sound feedback on events can be turned On and Off.

- You can let the operating system check events for you and you can act on the events when they happen.
  You can define a section inside which a jump to a specified label is performed when an event happens.
  You can define a list of events and when one of these events happens, the OS will jump and release the event checking.
  When an event has released the event checking state the actual releasing event can be read in a task local event register.
- You can send an event to the Scout via the IR-link and the Scout will act as if the event happened.

Scout access control

One of the basic problems to be solved in the Scout is the control of access to common output resources. From the RCX coding environment it has been seen, that it is very difficult to determine program flow when more than one task (running in parallel) are controlling the same output devices e.g. motors. The basic paradigm of access control is:

- One and only one task can have access to an **Output Resource** at the time.

In the Scout, this is done by defining **Access Control Sections**:

- An Access Control Section contains a block of commands using a set of output resources.
- An Access Control Section is characterized by a priority level, a set of output resources and a resume point.

Conflicts on access to resources are resolved by Section Priority.

Example:

A task is running in an Access Control Section with a certain priority level and involving Motor A control.
Another task enters an Access Control Section of same or higher priority level also involving Motor A control.

Result: Execution of the first task will be returned to the resume point of the section.
The interrupting task will start execution of its Motor A control program section.

Communicating with the Scout

Communicating with the Scout is done through the IR-link.

- Commands from an application through the IR Tower. Direct commands or downloads of programs.
  This is done through the Ghost optionally using Assembler, Block splitter or other Ghost utilities.
- Commands from the Remote control.
- PB messages from another Scout or an RCX.

An application can get any data from the Scout by uploading data blocks of variable size.

The Program Block Library

The Program Blocks Library is a collection of Program Blocks contained in subroutines. The Library is placed in ROM.

Each Task has a set of local variables. When a Task calls a Program Block (subroutine) it uses these local variables to set up the desired functionality of the Block.

From a program these blocks can be used as macro commands: By setting up local variables and calling the subroutine, a lot of functionality is achieved easily.

Installation

Simply extract all the files from the installation into the same directory. If desirable the target directory can be added to the PATH environment variable.

Package content

The two SDK downloads should consist of the following files:

ScoutSDK.pdf	The document you are reading now
ScoutTool.exe	An interactive Microsoft Windows application for programming and monitoring various aspects of the Scout. Great for experimentation.
ScoutTool.pdf	An online help document explaining how the ScoutTool.exe application works.
ScoutDOS.exe	A Microsoft DOS command line application for translating and downloading Scout byte code assembly programs and commands. Can be used as a back-end for other programming systems.
Lasm.dll	Assembler kernel
Scout.dll	Scout specific information
PbkComm32.dll	P-Brick Communication
PbkMouse.exe	A utility function to resolve conflicts between serial mice and IR Towers – used by PbkComm32.dll.
ScoutDef.h	Useful macros and definitions for general Scout byte code assembly programs. Most of the examples in this document assumes that the program contains a: #include “ScoutDef.h” directive or that the relevant #define macros are available.
“Samples” folder	This folder contains a few sample programs to help show some of the various capabilities and structure of LASM
Helloworld.txt	Simple LASM command to make the Scout play a sound
RepeatAfterMe.txt	Simple LASM program that plays 5 notes, one after the other
WatchYourStep.txt	Not-so-simple LASM program that implements event/sensor watchers for the timer and touch sensors

If files are missing, please check back at http://www.legomindstorms.com for updates or new information.

Getting started – ScoutDOS.exe

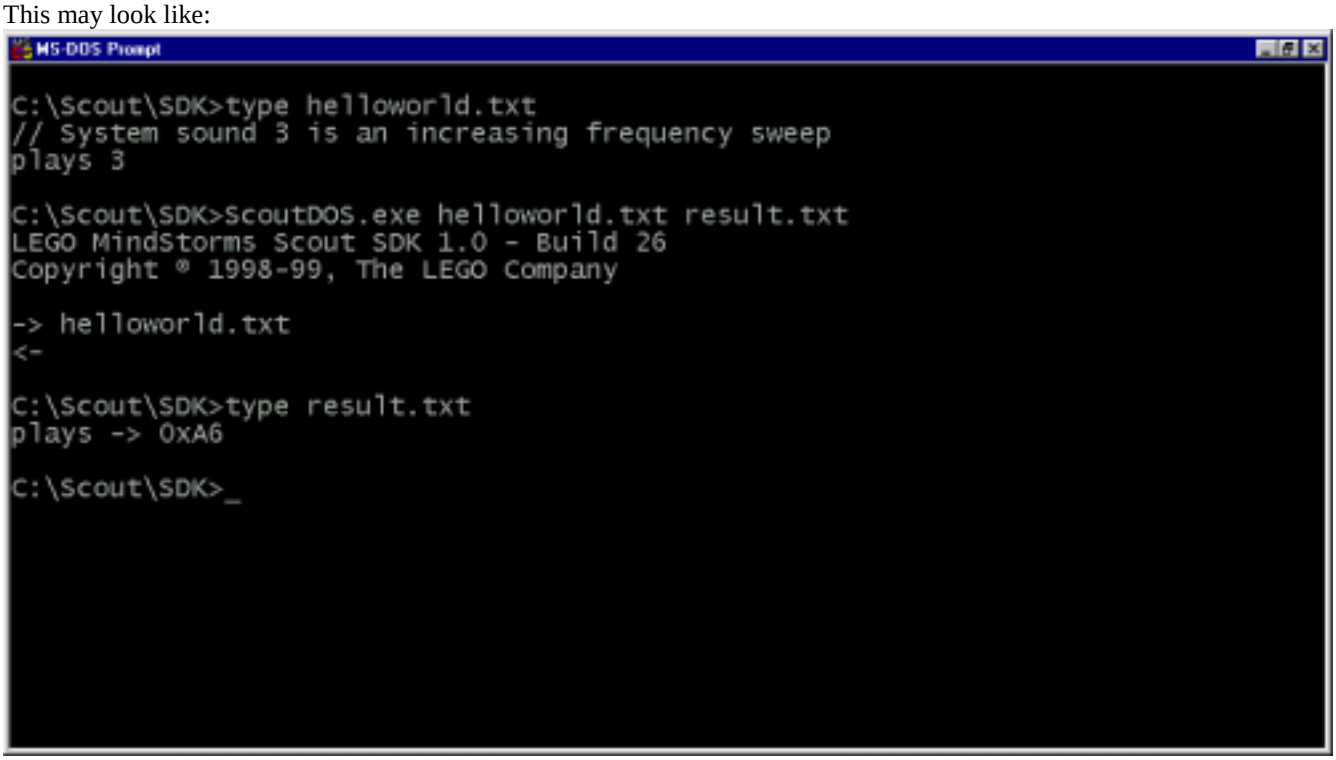
To get you started, some sample programs and download sessions are included below. They show you how to execute direct commands and how to download both a simple program and a not-so-simple sensor watcher program. For detailed descriptions of the commands see Instructions/opcodes.

Program arguments – example session

The ScoutDOS.exe program simply takes one or two filenames as parameters. The file is then compiled and, if no errors were found, downloaded. Status is output to the console window or DOS box ('stdout'). The result from executing direct commands may be output to the second optional file name and that file will then be overwritten.

To see the effect for yourself, run the ScoutDOS.exe program in an MS-DOS (Command) Prompt from the directory where you installed the files. Alternatively you can call the program (as a shell) with filename parameters from another program.

This may look like:



The exact display may change depending on your program files and installation directory.

Example programs

Hello World

The Scout equivalent of the famous 'C' Hello World program is to get the Scout to play a sound. The following example shows this (the program is so simple, you have to type it in yourself).

```
// System sound 3 is an ascending frequency sweep
plays   3
```

The effect of executing ScoutDOS.exe with the file should be an almost instant sound feedback from the Scout.

Repeat after me
The following program calls a subroutine five times to play the note ‘A’ raised an octave between each time.

```
#include “ScoutDef.h”

#define TASK_MAIN   0
#define SUB_PLAY    0

#define LOCAL_VAR_0 10
#define LOCAL_VAR_1 11

; This subroutine plays a note for half a second,
; the frequency is passed in a local variable
sub SUB_PLAY
  playv   LOCAL_VAR_0, FR_MS_500
  wait    SRC_CON, FR_MS_500 + FR_MS_50 ; the Scout has no sound buffer so put in
                                        ; waits to get the timing right.
ends

task TASK_MAIN
  setv    LOCAL_VAR_1, SRC_CON, 5       ; the number of iterations
  setv    LOCAL_VAR_0, SRC_CON, TONE_A5 ; the note 'A'
startloop_label:
  decvjn  LOCAL_VAR_1, endloop_label    ; decrement the loop variable and
                                        ; exit the loop if it becomes negative
  calls   SUB_PLAY
  mulv    LOCAL_VAR_0, SRC_CON, 2       ; doubling the frequency equals
                                        ; raising the note one octave
  jmp     startloop_label
endloop_label:
endt
```

The effect of downloading the program and pressing the ‘Run’ button should be that you hear the note ‘A’ played five times and raised an octave between each note (the last note may be hard to hear).

Watch your step
The following program sets up a timer to generate an event every second and then watches both touch sensors for being pressed. The timer generates a heart beat pulse, whereas the touch sensor watcher plays another sound when a sensor is pressed.

```
#include "ScoutDef.h"

#define TASK_MAIN   0
#define TASK_TOUCH  1
#define TASK_TIMER  2

#define LOCAL_VAR_0 10
#define LOCAL_VAR_1 11


task TASK_MAIN
  ; initialization
  out     OUT_OFF, OUTLIST_AB           ; turn motors off
  setfb   SRC_CON, FBMASK_NO_FB         ; shut the system up
  tmrs    0, SRC_CON, CR_SEC_1          ; wait a second

  ; start sensor watchers
  start   TASK_TOUCH
  start   TASK_TIMER
endt
```

```
task TASK_TOUCH
starttask_label:

  mone    SRC_CON, EVENT_TPR, watchercode_label

forever_label:
  jmp     forever_label                ; wait here until (one of) the event(s) happens

watchercode_label:

  plays   SND_BEEP

  jmp     starttask_label
endt


task TASK_TIMER
starttask_label:

  mone    SRC_CON, EVENT_TMR1, watchercode_label

forever_label:
  jmp     forever_label                ; wait here until the timer triggers

watchercode_label:

  plays   SND_CLICK

  jmp     starttask_label
endt
```

Instead of just playing sounds, the touch sensors could control motor power or something similar.

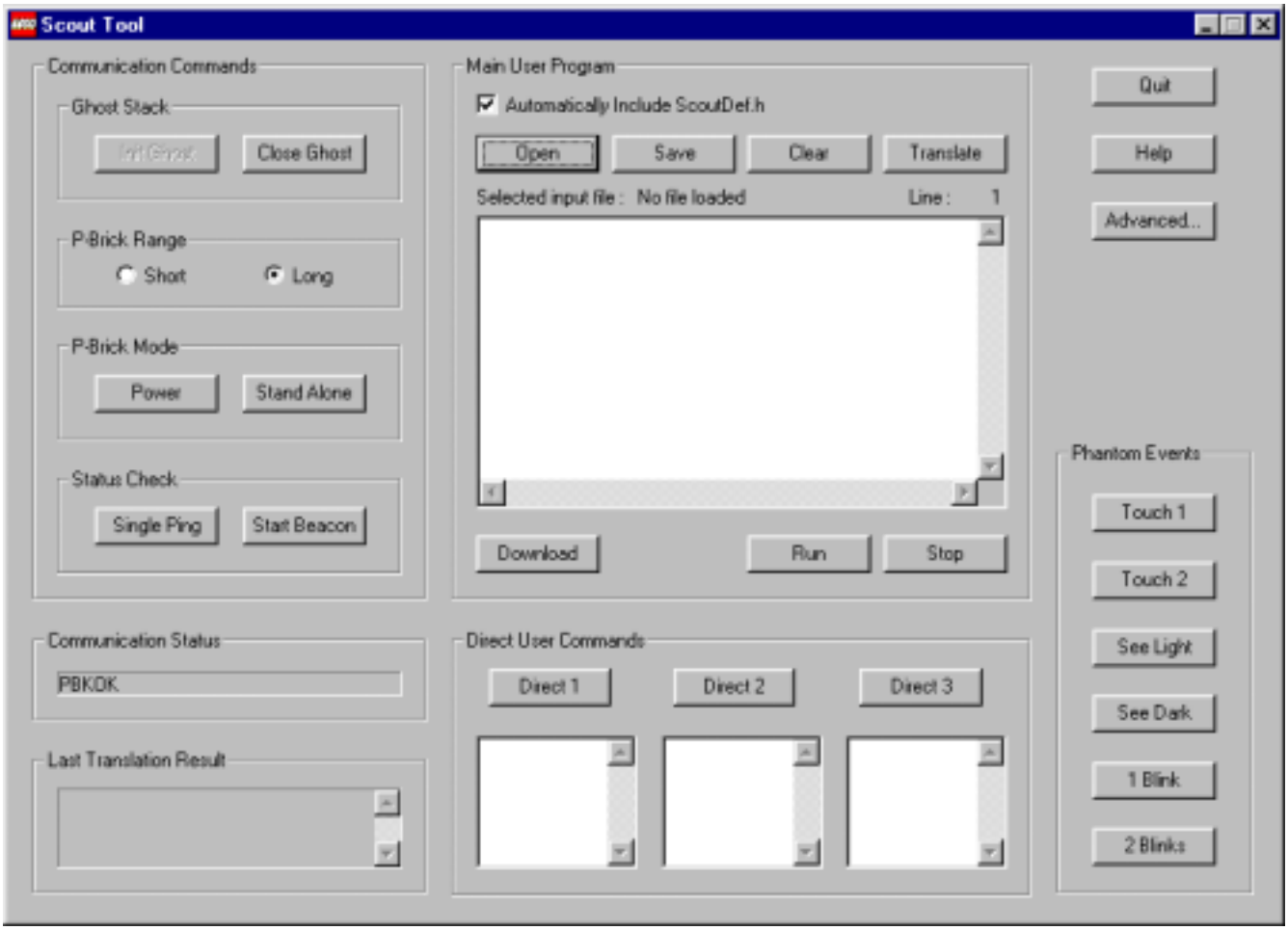
Remember to press the ‘Run’ button to actually start your program.

Getting started – ScoutTool.exe

The preceding chapter dealt with the command line application ScoutDOS.exe which is good for batch processing or for use as a back-end for another system. To help with experimentation and to support the program development and debugging process, another application, ScoutTool.exe, is supplied.

ScoutTool.exe provides an Interactive Development Environment (IDE) with a Graphical User Interface (GUI) and it is a 32-bit Microsoft Windows application.

The main screen looks like this:



There are a number of buttons to control the set-up of the Scout and the communication between the Scout and the PC.

The text boxes are used to write Scout byte code assembly programs, which can be downloaded and run.

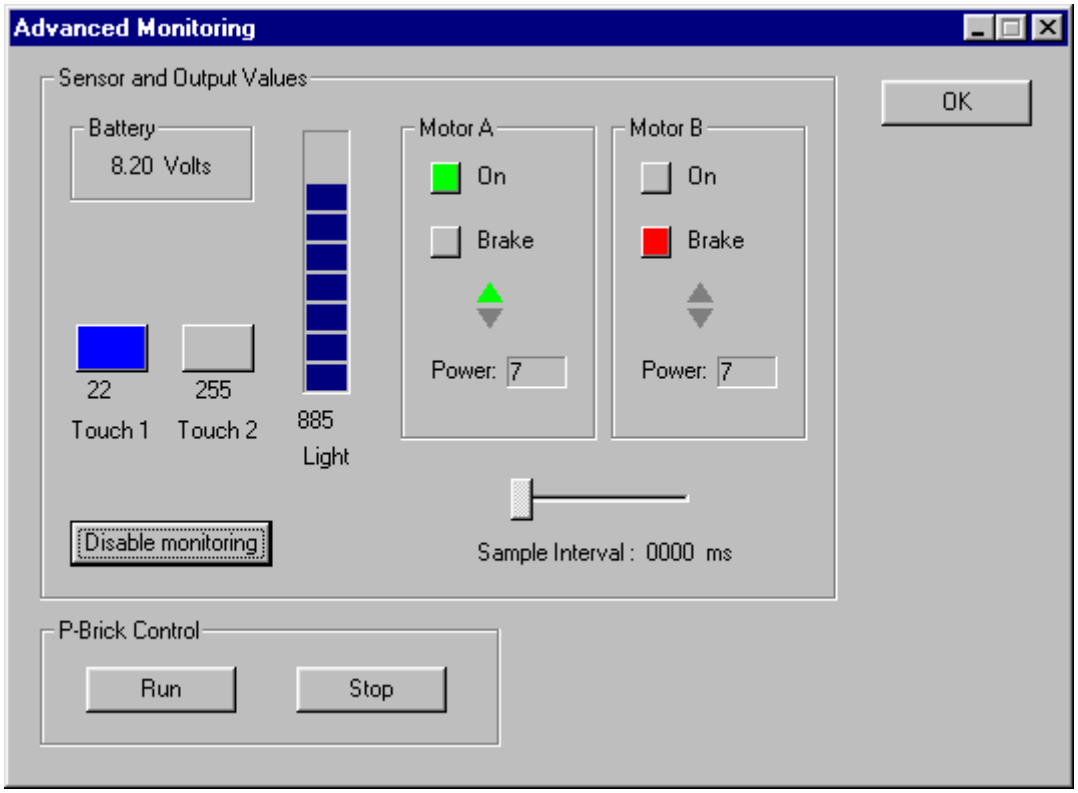
There are also facilities for writing small collections of user commands that are immediately run when the Direct User Commands buttons are pressed.

There are some status fields that show the result of translating the user programs or commands or the communication status.

Lastly, there are a collection of buttons to generate simulated events that the Scout can recognise and may react to.

Advanced Monitoring

By pressing the **Advanced** button you get to see the Advanced Monitoring screen:



You can start and stop the program and use the display to get dynamic feedback on what the program sees – this may help you to understand your programs behavior.

The ScoutTool.exe application is explained in more depth in the document ScoutTool.pdf.

Program syntax

The main elements of LEGO Byte Code Assembly (LASM) programs are described in the sections below.

Commands

A command consists of the opcode/mnemonic and a (possibly empty) list of parameters, such as:

```
                opcode      [arg1, …, argn]
```

Commands are separated by NEWLINE characters; i.e. there can be at most one command per line.

The parameters can be expressions using numerical operators such as '+', '-', '*', '/', bit operators such as '~', '&', '|', relational operators such as '<', '>', '<=', '>=', '==', '!=' and logical operators such as '&&' and '||'.

Please note that the translator is case sensitive and that all commands use lower case.

Labels

Labels are symbolic program addresses that can only be used within the same structure they are defined in. Labels are not allowed together with immediate commands. Since no immediate commands use labels, their use is flagged as explicit errors rather than safely ignoring them.

A label is an alphanumeric string at the beginning of a new line immediately followed by a colon, as in:

```
MyLoopLabel:    opcode      [arg1, …, argn]
```

Labels may be indented with space and TAB characters.

Comments

Everything on a line after a semicolon (';') or a C++ style comment header ('//') is treated as a comment, as in:

```
MyLoopLabel:    opcode      [arg1, …, argn]     ; this is where my loop starts
```

For convenience, 'C' style comments ("/* … */") are also allowed.

Structures

In the programs, tasks and subroutines are used as the main structuring units. The structures begin with 'task id' or 'sub id' and are terminated by 'endt' and 'ends' respectively.

Everything outside the scope of a task or a subroutine (a structure in general) is considered a direct command.

```
                opcode1     arg1        ; a direct command

                task        0           ; task 0 is the main task

MyLoopLabel:    opcode2     arg1, …     ; this is where my loop starts
                                        ; this is inside a structure,
                                        ; so labels are allowed
                …
                endt                    ; end of main task

                sub         2           ; a user subroutine
                …
                ends
```

Pre-processor directives

To improve the structure and readability of the source files, a few standard 'C' type pre-processor directives will be supported:

```
// Useful symbolic names for command argument SOURCE.
#define SRC_VAR      0
#define SRC_CON      2

#define MASK   0xFFFF                    // useful for 16 bit values
#define VALUE  MASK & 123456             // macros can contain expressions

// Either some common code/sub/task or
// common definitions like above.
#include "usefulstuff.asm"

andv    array_base + 3, SRC_CON, VALUE   // use the macros in commands
```

The #define macros do not currently accept parameters. There is a nesting limit of 16 with #include and #define macro expansions.

By using the '//' comment format in include files, useful definitions can be shared between assembler and C/C++ programs as header include files.

Mixing programs with direct commands

The ScoutDOS.exe program can also execute direct commands, and it is possible to mix program elements (tasks and subroutines) with direct commands so that one can set up a complete system from within the same program.

Parameters

In the assembler command list below, the following abbreviations are used:

Abbreviation	Explanation
'src', 's1', 's2'	The source of a number, i.e. its type or origin
'val', 'v1', 'v2'	The 'value' of a number, i.e. what element of the type that its source tells
'number', 'n1', 'n2', 'n3', 'n4','n5',	A direct number
'onoff'	A Boolean expression
'motors'	A bit field list of the affected motors
'eventlist'	A bit field list of the relevant events
'resources'	A bit field list of the accessed resources
'commands'	A bit field list of the requested remote commands
'label', 'relative address', 'offset', 'adr'	All addresses in commands that can jump to different program instructions are relative to the address field inside the byte code command. They can either be specified directly as a (signed) number or by reference to a label. The assembler will then compute the offset, when it knows the addresses of both the label and the program instruction. Offsets come in two sizes: **short** (-128 to 127) and **long** (-32768 to 32767). Using labels is by far the safest option, as the assembler knows about the necessary field offsets and address calculations. Besides, carefully named labels can help in documenting the program.
'relop'	Relational operator for comparisons: 0 (greater than), 1 (less than), 2 (equal to) and 3 (different from).

For the exact bit masks see the last section in this chapter.

Instructions/opcodes

The following section lists, in alphabetical order, all the byte code assembly commands for the Scout brick and the legal parameter sources. Legal ranges for each source type are listed later in the document.

Opcode 'absv'
Sets variable 'number' with the absolute value of the given value

```
absv    number, src, val
```

Legal range for 'src': 0 (variables) or 2 (constants).

Opcode 'andv'
Sets variable 'number' with the result of the bit wise AND of the given value and variable 'number'

```
andv    number, src, val
```

Legal range for 'src': 0 (variables) or 2 (constants).

Opcode 'boot'
Starts the Power mode command interpreter in the P-Brick, if the string matches the handshake. Only possible as a direct command.

```
boot    n1, n2, n3, n4, n5
```

The handshake numbers are "LEGO®" as ASCII values (0x4C, 0x45, 0x47, 0x4F, 0xAE).

Opcode 'calls'
Executes subroutine 'number'. The firmware does not support subroutines to call other subroutines. Only possible as a program command.

```
calls   number
```

Legal range for 'number': 0-32 (0-2 is user subroutines, 3-32 is the built-in system subroutines - see "Program Block Library").

Opcode 'chk'
Checks the condition and jumps to 'adr' (short offset) if the condition evaluates to FALSE. Only possible as a program command.

```
chk     s1, v1, relop, s2, v2, short adr
```

Legal range for 's1' and 's2': 0 (variables), 1 (timers), 2 (constants), 3 (motor status), 9 (sensor value), 12 (raw sensor value), 15 (IR message), 17 (output setup), 18 (stand alone setup), 21 (counter), 23 (task event register) and 24 (event sound feedback register). Only the first parameter source/value pair can indicate a constant value (2).
Legal range for 'relop': 0 (greater than), 1 (less than), 2 (equal to) and 3 (not equal to).

Opcode 'chkl'

```
chkl    s1, v1, relop, s2, v2, long adr
```

As above, but with a long relative address offset

Opcode 'cntd'
Decrements one of the built-in counters.

```
cntd    number
```

Legal range for 'number': 0-1.

Opcode 'cnti'
Increments one of the built-in counters.

```
cnti    number
```

Legal range for 'number': 0-1.

Opcode 'cnts'
Sets the counter value (for overflow detection and event generation).

```
cnts    number, src, val
```

Legal range for 'number': 0-1.
Legal range for 'src': 0 (variable), 2 (constant value) and 4 (random value)

Opcode 'cntz'
Clears the given counter.

```
cntz    number
```

Legal range for 'number': 0-1.

Opcode 'decvjn'
Decrements the (loop) variable 'number' and jumps if the value becomes less than zero (negative). Only possible as a program command.

```
decvjn  number, relative address
```

Legal range for 'number': 0-17 (0-9 are global variables and 10-17 are local variables to the task).

Opcode 'decvjnl'

```
decvjnl number, relative address
```

As above, but with a long relative address offset.

Opcode 'dels'
Deletes one or all subroutines. Only possible as a direct command.

```
dels    [number]
```

Legal range for 'number': 0-2.

Opcode ‘delt‘
Deletes one or all tasks. Only possible as a direct command.

```
delt    [number]
```

Legal range for ‘number’: 0-5.

Opcode ‘dir’
Changes the direction of the listed outputs.

```
dir     action, motors
```

Legal range for ‘action’: 0 (backward), 1 (change direction), 2 (forward).
Legal range for ‘motors’: 1-7 (a bit mask).

Opcode ‘divv’
Divides variable ‘number’ with the given value

```
divv    number, src, val
```

Legal range for ‘src’: 0 (variables), 2 (constants).

Opcode ‘event’
Makes the Scout behave as if one or more events had occurred

```
event   eventlist
```

Legal range for ‘eventlist’: 1-32767 (a bit mask for the 15 possible system events).

Opcode ‘gdir’
Changes the global direction settings of the listed outputs, so that all subsequent normal motor direction commands will be overridden:

```
gdir    action, motors
```

Legal range for ‘action’: 0 (backward), 1 (change direction), 2 (forward).
Legal range for ‘motors’: 1-7 (a bit mask).

Opcode ‘gout’
Changes the master output status of the listed outputs, so that all subsequent normal motor power commands will be overridden.

```
gout    action, motors
```

Legal range for ‘action’: 0 (float), 1 (off), 2 (on).
Legal range for ‘motors’ is a bit-mask: 1-7.

Opcode 'gpwr'
Master output power instructions for motors, so that all subsequent normal motor power commands will be overridden.

```
gpwr    motors, source, number
```

Legal range for 'source': 0 (variable), 2 (constant value) and 4 (random number) – all limited to 0-7
Legal range for 'motors': 1-7 (a bit mask)

Opcode 'jmp'
Go to the given address (short offset) and continue the program from there. Only possible as a program command.

```
jmp     short relative address
```

Opcode 'jmpl'
As 'jmp' but with long offset.

```
jmpl    long relative address
```

To minimize program size, try to use the short form and only when the translation complains, change to the long form.

Opcode 'light'
Turns the VLL output (the red LED) on or off for decorative purposes

```
light   onoff
```

Legal range for 'onoff': 0-1 (Boolean).

Opcode 'lsbt'
Sets the Light Sensor Blink Time

```
lsbt    src, val
```

Legal range for 'src': 0 (variables) and 2 (constant values).
Legal range for 'val': 1-32767 (measured in 0.01 s)

Opcode 'lscal'
Uses ambient light levels for setting up the light sensor trigger levels. This command sets upper and lower thresholds and the hysteresis for the light sensor.

```
lscal
```

This command takes no parameters.

Opcode 'lsh'
Sets the Light Sensor hysteresis

```
lsh     src, val
```

Legal range for 'src': 0 (variables) and 2 (constant values).
Legal range for 'val': 0-1020.

Opcode 'lslt'
Sets the Light Sensor low threshold

```
lslt    src, val
```

Legal range for 'src': 0 (variables) and 2 (constant values).
Legal range for 'val': 0-1020 (a low value indicates a bright environment).

Opcode 'lsut'
Sets the Light Sensor upper threshold

```
lsut    src, val
```

Legal range for 'src': 0 (variables) and 2 (constant values).
Legal range for 'val': 0-1020 (a low value indicates a bright environment).

Opcode 'monal'
Tries to grab the listed resources at the requested priority or jumps to address if unsuccessful. Also jumps to address if pre-empted later on. Only possible as a program command.

```
monal   resources, address
```

Legal range for 'resources': 0x01-0x0F (a bit mask).

Opcode 'monax'
Stops monitoring the (last) set of resources. Only possible as a program command.

```
monax
```

This command takes no parameters.

Opcode 'mone'
Sets the event list to interrupt normal execution flow and jump to address 'label' on detection. Only possible as a program command.

```
mone    src, val, label
```

Legal range for 'src': 0 (variable), 2 (constant value) and 4 (random number).

Opcode 'monel'
As 'mone' but with long offset

```
monel   src, val, label
```

To minimize program size, try to use the short form and only when the translation complains, change to the long form.

Opcode 'monex'
Stops event monitoring. Only possible as a program command.

```
monex
```

This command takes no parameters.

Opcode 'msg'
Sends the given value as an 8-bit P-Brick message

```
msg     src, number
```

Legal range for 'src': 0 (variable) and 2 (constant value).

Opcode 'msgs'
Sets the P-Brick message (buffer) by mimicking receipt of a message that also generates an event. Only possible as a direct command.

```
msgs    number
```

Legal range for 'number': 0-255.

Opcode 'msgz'
Clears the P-Brick message (buffer)

```
msgz
```

This command takes no parameters.

Opcode 'mulv'
Multiplies variable 'number' with the given value

```
mulv    number, src, val
```

Legal range for 'src': 0 (variable) and 2 (constant value).

Opcode 'offp'
Turns the P-Brick off

```
offp
```

This command takes no parameters.

Opcode 'orv'
Sets variable 'number' with the result of the bit wise OR of the given value and variable 'number'

```
orv     number, src, val
```

Legal range for 'src': 0 (variable) and 2 (constant value).

Opcode 'out'
Changes the status of the listed outputs.

```
out     action, motors
```

Legal range for 'action': 0 (float), 1 (off) and 2 (on).
Legal range for 'motors': 1-7 (a bit mask).

Opcode 'ping'
Checks that a P-Brick is available. Only possible as a direct command.

```
ping
```

This command takes no parameters.

Opcode 'plays'
Plays a given system sound

```
plays   number
```

Legal range for 'number': 0-27.

Opcode 'playt'
Plays a tone with a given frequency for a given duration

```
playt   freq, duration
```

Legal range for 'freq': 30-20000 (Hz).
Legal range for 'duration': 1-255 (measured in 0.01 s)

Opcode 'playv'
Plays a tone with a given frequency (read from variable 'number') for a given duration. This is useful for playing music because a variable may be manipulated in many ways (such as being multiplied by 2) before being played again.

```
playv   number, duration
```

Legal range for 'duration': 1-255 (measured in 0.01 s)

Opcode 'pollm'
Retrieves a memory snapshot. Only possible as a direct command.

```
pollm   adr, size
```

Legal range for 'adr': 0x0040-0x0440.
Legal range for 'size': 1-150.
'adr' + 'size' cannot exceed 0x0440.

Opcode 'pollp'
Retrieves the ROM and Firmware RAM versions if the magic numbers are correct. Only possible as a direct command.

```
pollp   n1, n2, n3, n4, n5
```

The magic numbers are: 1, 3, 5, 7, and 11.

Opcode 'pwr'
Sets the power level for the listed outputs

```
pwr     motors, src, val
```

Legal range for 'motors': 1-7 (a bit mask).
Legal range for 'src': 0 (variable), 2 (constant value) and 4 (random number).

Opcode 'remote'
Sends the same remote commands as the buttons on the hand held device. The Scout does not reply to remote commands. Remote motor commands time out after 125 ms, so the remote control repeatedly sends the keys that the user is pressing. When the keys are released, a special remote command is sent to allow the non-motor related commands to be handled again (those that don't repeat their effect). To mimic that behavior from the PC, it is necessary to interleave such key release commands. Only possible as a direct command.

```
remote  commands
```

Legal range for 'commands': 0x0000-0xFFFF (see "Remote commands" for a description of the bits).
All buttons correspond to a single bit in the 16-bit 'commands' value. The motor commands can be combined with the other commands (but only one forward or reverse command per motor).

Opcode 'rules'
Selects Scout Motion, Touch, Light, Time and FX rules for the Scout Stand Alone mode.

```
rules   motion, touch, light, time, fx
```

See "Inside the Scout: Basic functionality" for the legal ranges of the various rule groups.

Opcode 'scout'
Selects Stand Alone (SA) or Power mode

```
scout   number
```

Legal range for 'number': 0 (Stand Alone mode) and 1 (Power mode).

Opcode 'setfb'
Selects which (external) events should result in a system sound being played

```
setfb   src, val
```

Legal sources are: 0 (variable), 2 (constant value) and 4 (random number).
The resulting value has the same structure as an event list with each bit corresponding to a system event.

Opcode 'setp'
Sets the task priority to be used for access control. Only possible as a program command.

```
setp    number
```

Legal range for 'number': 0-7 (with 0 being the most important/highest priority).

Opcode 'setv'
Sets variable 'number' to the given value

```
setv    number, src, val
```

Legal range for 'src': 0 (variables), 1 (timers), 2 (constants), 3 (motor status), 4 (random number), 9 (sensor value), 10 (sensor type), 12 (raw sensor value), 15 (IR message), 17 (output setup), 18 (stand alone setup), 21 (counter), 23 (task event register) and 24 (event sound feedback register).

Opcode 'sgnv'
Sets variable 'number' with the result of the sign test of the given value

```
sgnv    number, src, val
```

Legal range for 'src': 0 (variable) and 2 (constant value).

Opcode 'sound'
Controls global sound settings (allows a 'mute' functionality) and selects which scheme is currently used for system sounds 10-27:

```
sound   sound_enable, sound_onoff, soundset_number
```

Legal range for 'sound_enable': 0 (disregard 'sound_onoff' and select 'soundset_number' for the system sounds) and 1 (disregard 'soundset_number' and use 'sound_onoff' to globally control all sounds).
Legal range for 'sound_onoff': 0 (mute all sounds) and 1 (allow sounds to pass through).
Legal range for 'soundset_number': 0 (NoSoundset), 1 (Basic), 2 (Bug), 3 (Alarm), 4 (Random) and 5 (Science)

Opcode 'start'
Starts executing a task from the beginning or restarts it if it was already running

```
start   number
```

Legal range for 'number': 0-5.

Opcode 'stop'
Stops execution of one or all tasks

```
stop    [number]
```

Legal range for 'number': 0-5.

Opcode 'subv'
Subtracts the given value from variable 'number'

```
subv    number, src, val
```

Legal range for 'src': 0 (variable) and 2 (constant value).

Opcode 'sumv'
Adds the given value to variable 'number'

```
sumv    number, src, val
```

Legal range for 'src': 0 (variable) and 2 (constant value).

Opcode 'tmrs'
Sets the timer limit (for overflow/wrap-around detection and event generation)

```
tmrs    number, src, val
```

Legal range for 'src': 0 (variable), 2 (constant value) and 4 (random number).

Opcode 'tmrz'
Clears the given timers

```
tmrz    number
```

Legal range for 'number': 0-2.

Opcode 'tout'
Sets the power down time (time-out) in minutes.

```
tout    time
```

Legal range for 'time': 0-255 (0 means never).

Opcode 'txs'
Sets the P-Brick transmit power level.

```
txs     range
```

Legal range for 'range': 0 (low level/short range), 1 (high level/long range).

Opcode 'vll'
Sends a 7-bit VLL command out over the VLL output

```
vll     source, number
```

Legal range for 'source': 0 (variable) and 2 (constant value).

Opcode 'wait'
Pauses the execution of the task for a given number of 10 ms. Only possible as a program command.

```
wait    src, val
```

Legal range for 'src': 0 (variable), 2 (constant value) and 4 (random number).

Virtual machine specifics

The Scout has the following characteristics:

Sources

Source	Item	Value	Explanation
0	Variables	0-9 10-17	Variables 0-9 are shared global variables Variables 10-17 are local to each task and the sub routines it may call. This allows safe parameter passing to sub routines if it refers to the passed local variables. It also means that commands involving variables have different legal ranges depending on whether they are direct or program commands.
1	Timers	0-3	Timers are free running global counters with a resolution of 100 ms (i.e. 10 ticks per second). When reset, they immediately start running (again).
2	Constants	–32768 to +32767	Usually
3	Output Status Register	0-2	Bit 0-2: Power Bit 3-3: Direction; 1 – Forward, 0 – Reverse Bit 4-5: Output no. Bit 6-6: 1- Break, 0 – Float Bit 7-7: 1 – On, 0 – Off
4	Random	1-32767	
9	Sensor Value	0-2	This register contains the processed sensor value.
10	Sensor Type		This register contains the sensor type 1 = Normal Touch Sensor 5 = ID0 Touch Sensor (Yellow) 6 = ID1 Touch Sensor (Red) 7 = ID2 Touch Sensor (White)
12	Sensor Raw		This register contains the raw 8 bit A/D converted value
15	PB Message	0	
17	Output Setup Register	0-2	See source 3
18	Stand Alone Setup	0-4	Contains the rule selection for the five groups: 0 – Motion 1 – Touch 2 – Light 3 – Time 4 – FX
19	Light Sensor Parameters	0-3	Contains the light sensor control parameters 0 – Upper Threshold 1 – Lower Threshold 2 – Hysteresis 3 – Blink Time The register is write-only – it cannot be queried by user programs.

Source	Item	Value	Explanation
20	Timer Limit	0-2	Timers generate events when they reach their limit. **After reaching the limit, a timer is automatically reset and without a set limit a timer does not run at all.** Setting a timer limit also automatically resets the timer. To achieve RCX like timer behavior, the limit should be set to the maximum value (32767).
21	Counters	0-1	Counters are special to the Scout and behave in many ways as special global variables, with the limitation that they can only be reset (to zero), incremented and decremented.
22	Counter Limit		Counters generate events when they reach their limit. They can act as score keepers in some competitive applications where a task can monitor the counter limit as the end of the game, or a timer can generate time-out warnings.
23	Task Event Register	0-5	Each task gets a copy of the relevant bits in the global event register, when monitored events occur. The bits are thus: 0x0001 == Touch 1 Pressed 0x0002 == Touch 1 Released 0x0004 == Touch 2 Pressed 0x0008 == Touch 2 Released 0x0010 == Light Sensor Enter Light State 0x0020 == Light Sensor Enter Normal State 0x0040 == Light Sensor Enter Dark State 0x0080 == Light Sensor 1 Light Blink 0x0100 == Light Sensor 2 Light Blinks 0x0200 == Counter 0 Over Limit 0x0400 == Counter 1 Over Limit 0x0800 == Timer 0 Over Limit 0x1000 == Timer 1 Over Limit 0x2000 == Timer 2 Over Limit 0x4000 == PBMessage Received
24	Event Sound Feedback Register	0	Contains a bit for each of the system events above, indicating whether or not that event should generate the corresponding system sound when it occurs.

Output resources

The following table shows how shared output resources are constructed.

Bits	Explanation
0x01	MOTOR_A
0x02	MOTOR_B
0x04	SOUND
0x08	VLL_OUT

For multiple access, the resources can be OR’ed together.

Remote commands

The following table shows how remote control messages are constructed.

Bits	Explanation
0x0000	Key(s) released
0x0100	PBMessage 1
0x0200	PBMessage 2
0x0400	PBMessage 3
0x0800	Turn motor A on in forward direction
0x1000	Turn motor B on in forward direction
0x2000	Turn motor C on in forward direction
0x4000	Turn motor A on in backwards direction
0x8000	Turn motor B on in backwards direction
0x0001	Turn motor C on in backwards direction
0x0002	Select Program 1
0x0004	Select Program 2
0x0008	Select Program 3
0x0010	Select Program 4
0x0020	Select Program 5
0x0040	Stop the program and turn all motors off
0x0080	Play a sound

The _OUTPUT_ commands can be OR’ed with the other commands (but only one command per output port).

Assembly program structure templates

In order to produce structured programs even in assembly the following templates are supplied as a programming aid like the structuring offered by SPIRIT.OCX. The examples start with a SPIRIT.OCX style code snippet using 'PB' as the object name. If you do not know about SPIRIT.OCX, you can look at www.legomindstorms.com/sdk to see and download another Software Developers Kit for SPIRIT.OCX. The SPIRIT.OCX SDK is intended for more detailed programming of the LEGO MindStorms Robotics Invention System 'RCX' programmable brick and the LEGO Technic CyberMaster programmable brick.

The program templates do not distinguish between short/long forms of the commands, where they exist. The smallest programs are achieved by using the short form initially and then changing it to the long form if the assembler complains that the relative address is out of range.

IF … ENDIF

Structures like:

```
PB.If s1, v1, relop, s2, v2
       true code
PB.EndIf
```

Gets implemented as:

```
       chk    s1, v1, relop, s2, v2, endiflabel
              {true code}
endiflabel:
```

IF … ELSE … ENDIF

Structures like:

```
PB.If s1, v1, relop, s2, v2
       true code
PB.Else
       false code
PB.EndIf
```

Gets implemented as:

```
       chk    s1, v1, relop, s2, v2, elseiflabel
              {true code}
       jmp    endiflabel
elseiflabel:
              {false code}
endiflabel:
```

WHILE … ENDWHILE

Structures like:

```
PB.While s1, v1, relop, s2, v2
       while code
PB.EndWhile
```

Gets implemented as:

```
startwhilelabel:
        chk     s1, v1, relop, s2, v2, endwhilelabel
                {while code}
        jmp     startwhilelabel
endwhilelabel:
```

DO … WHILE

A structure like:

```
        PB.Do
               while code
        PB.While s1, v1, relop, s2, v2
```

While not available in SPIRIT.OCX it would be implemented as:

```
startwhilelabel:
                {while code}
        chk     s1, v1, relop, s2, v2, endwhilelabel
        jmp     startwhilelabel
endwhilelabel:
```

DO … UNTIL

Structures like:

```
        PB.Do
               until code
        PB.Until s1, v1, relop, s2, v2
```

While not available in SPIRIT.OCX it would be implemented as:

```
startuntillabel:
                {until code}
        chk     s1, v1, opposite relop, s2, v2, enduntillabel
        jmp     startuntillabel
enduntillabel:
```

The opposite of ‘<’ is ‘>’ and the opposite of ‘==’ is ‘!=’ and vice versa. An UNTIL loop executes until a condition is met which can be useful sometimes.

FOREVER … ENDLOOP

Structures like:

```
        PB.Loop 2, 0 ‘ forever
               loop code
        PB.EndLoop
```

Gets implemented as:

```
startforeverlabel:
                {loop code}
        jmp     startforeverlabel
```

LOOP … ENDLOOP

Structures like:

```
PB.Loop src, val
       loop code
PB.EndLoop
```

Gets implemented as:

```
       setv   loopvarcounter, src, val
startlooplabel:
       decvjn loopvarcounter, endlooplabel
              {loop code}
       jmp    startlooplabel
endlooplabel:
```

SWITCH … CASE … ENDSWITCH

Structures like:

```
PB.Switch src, val
PB.Case val_1
       case 1 code
       break
…
PB.Case val_n
       case n code
       break
PB.Default
       default code
PB.EndSwitch
```

While not available in SPIRIT.OCX it would be implemented as:

```
case1check:
       chk    SRC_CON, val_1, EQ, src, val, case2check
              {case 1 code}
       jmp    endswitchlabel
case2check:
       …
casencheck:
       chk    SRC_CON, val_n, EQ, src, val, defaultcheck
              {case n code}
       jmp endswitchlabel
defaultcheck:                       ; no check – always true
              {default code}
endswitchlabel:
```

If fall-through behavior is wanted for some branches, one simply removes the ‘jmp endswitchlabel’ in the branch code.

If the case values are ordered numerically and sequentially (or can be brought to be so) a more program space- and run time-efficient scheme exists. It uses the **decvjn** byte code.

```
PB.Switch src, val
PB.Case 0:
       case 0 code
       break
…
PB.Case N:
       case n code
       break
PB.Default:
       default code
PB.EndSwitch
```

While not available in SPIRIT.OCX it would be implemented as:

```
       setv   casevar, src, val
       decvjn casevar, case0code
       …
       decvjn casevar, caseNcode
       jmp    defaultcode

case0code:
       {case n code}
       setv    casevar, SRC_CON, 0; restore case variable
       jmp endswitchlabel
       …
caseNcode:
       {case n code}
       setv    casevar, SRC_CON, N; restore case variable
       jmp endswitchlabel
defaultcode:                        ; no check – always true
       {default code}
       setv    casevar, SRC_CON, N; restore case variable
endswitchlabel:
```

If fall-through behavior is wanted for some branches, one simply removes the ‘jmp endswitchlabel’ commands.

Since the case works by decrementing the variable it may be necessary to reload the variable when exiting. This will not work with fall-through behavior, so using a separate variable for the switch statement expression is the best solution – it also saves even more program space and run time.

ENTER EVENT CHECK … EXIT EVENT CHECK

Structures like:

```
PB.EnterEventCheck  src, val ‘ eventlist
       non event code
PB.ExitEventCheck
```

While not available in SPIRIT.OCX it would be implemented as:

```
       mone   eventlist, exitlabel
starteventchecklabel:
       {loop code}
       jmp    starteventchecklabel
exitlabel:
       monex
```

WAIT UNTIL EVENT

Structures like:

```
        PB.WaitUntilEvent   src, val ‘ eventlist
```

While not available in SPIRIT.OCX it would be implemented as:

```
        mone   src, val, there
here:   jmp    here
there:
```

ENTER ACCESS CONTROL … EXIT ACCESS CONTROL

The access control mechanism is not using a nesting approach, so one can change the access control during the program execution. The exit access control command will remove all access control settings for that task.

A typical resume example can have the form:

```
resumelabel:
                {resume or initialization code or none}
        setp    priority
        mona    resources, resumelabel
                {application code using the listed resources}
        monax
```

If the initialization code is empty, then this is a busy-wait for one or more resources. An abort on contention strategy can have the form:

```
                {resume or initialization code or none}
        setp    priority
        mona    resources, abortlabel
                {application code using the listed resources}
        monax
                {possibly a jump over the abort code}
abortlabel:
                {clean up code}
```

The access controls can be nested or applied sequentially (since the exit command exits all access control).

SEMAPHORE BASED GUARDED ACCESS

By devoting a global variable to a semaphore (access guard), it is possible to provide secure and non-interruptible access to shared variables, which the access control monitor does not cover.

```
sub sGetSema
        subv    vSema, SRC_CON, 1          ; try to get the semaphore

checksemalabel:                            ; test to see if the task got the semaphore
        chk     SRC_CON, 0, GT, SRC_VAR, vSema, gotsemalabel

                sumv    vSema, SRC_CON, 1   ; no success, so give it back
                wait    SRC_RAN, FR_MS_100  ; wait randomly to prevent race-conditions
                                            ; on average, one will wait 50 ms.
                subv    vSema, SRC_CON, 1   ; then try to get the semaphore again

        jmp     checksemalabel

gotsemalabel:
ends
```

```
sub sReleaseSema
        sumv   vSema, SRC_CON, 1
ends
```

The semaphore is used in the following manner:

```
        calls  sGetSema
               {guarded code}
        calls  sReleaseSema
```

The program then waits in the first subroutine call until it gets the semaphore.

The three subroutines 29-31 implement this busy-wait semaphore functionality using global variables 0-2. The release part must be programmed directly.

TIMEOUT

In order to timeout one or more events one needs a timer. If you also want to measure the time spent waiting for the event(s), you need an extra timer or an extra variable.

Measuring timeout with an extra variable

```
        tmrs   TimeoutTimer, Src, Value   ; say when
        tmrz   TimeoutTimer               ; start now

        mone   SRC_CON, Events | TimeoutTimerOverLimitEvent, EvOrToutLabel

NotYetEvOrToutLabel:
        setv   TimeMeasureVar, SRC_TIMER, TimeoutTimer
        jmp    NotYetEvOrToutLabel

EvOrToutLabel:
        ; check to see if the event happened or it was a timeout
        setv   EventRegVar, SRC_EVENT, REG_TASKEVENT
        andv   EventRegVar, SRC_VAR, Events     ; mask out the relevant events

                                        ; exit if zero i.e. no events
        chk    SRC_CON, 0, NE, SRC_VAR, EventVarReg, ToutLabel

EvLabel:
               {event handling}
ToutLabel:
```

Measuring timeout with an extra timer

```
        tmrs   TimeoutTimer, Src, Value   ; say when
        tmrz   TimeoutTimer               ; start now

        tmrs   TimeoutMeasureTimer, SRC_CON, 32767
        tmrz   TimeoutMeasureTimer

        mone   SRC_CON, Events | TimeoutTimerOverLimitEvent, there
here:   jmp    here
there:

        setv   TimeMeasureVar, SRC_TIMER, TimeoutMeasureTimer

EvOrToutLabel:
               {as above}
```

Since timers are global, it may be better to use a local variable, especially since writing to the variable during the wait period is 'free' if the program is not to do other important stuff during the wait. Also if you want to use the actual measured period for later control, you will have to use an extra variable anyway.

Timeout without all the fuss
If the actual event, timeout or otherwise, is unimportant (the program just moves on to the next step), then a much simpler program is possible:

```
        tmrs   TimeoutTimer, Src, Value   ; say when
        tmrz   TimeoutTimer               ; start now

        mone   SRC_CON, Events | TimeoutTimerOverLimitEvent, there
here:   jmp    here
there:
```

This program will simply wait until an event or the special timeout event happens, before moving on.

Timeout without timers
An even simpler approach is possible that does not use program timers at all:

```
        mone   SRC_CON, Events, Handler
        wait   Src, Value                 ; wait for timeout here
        monex                             ; timeout expired
        jmp    Skip
Handler:
        …
Skip:
```

This program will simply abandon event monitoring on time-out and skip the event handling code.

General robotics programming topics

There are a number of general areas that one needs to understand in order to control robotic inventions by means of downloaded programs.

In general terms a robotic system is an invention that tries to achieve some goal by controlling actuators attached to output ports while reading and reacting to sensors attached to input ports.

The main areas of interest then becomes:
1. Controlling outputs, typically motors but also sound and light units.
2. Reading and processing inputs.
3. Reacting to external events (as seen through the input sensors).
4. Providing a sensible program structure that will manage all of the above, while possibly meeting an overall goal.

The LEGO P-Bricks are multi-tasking which means that they can execute a number of individual and/or separate jobs in parallel.

The preceding chapter has a number of examples of general structures (templates) to control the program flow.

Variables

Most programs require you to store and manipulate information for shorter or longer periods of time. In the LEGO P-Bricks you can do that with variables.

Variables do not have to be allocated – there exist a fixed number of variables and the program has to decide which variables to use for what.

The program examples below show various uses of variables

```
#include “ScoutDef.h”

#define GLOBAL_VAR   1                  ; a global variable for
                                        ; sharing data between tasks

#define LOCAL_VAR    10                 ; a task-local variable

        setv   GLOBAL_VAR, SRC_CON, 42    ; the ultimate answer

                                        ; make a local copy
        setv   LOCAL_VAR, SRC_VAR, GLOBAL_VAR
        sumv   LOCAL_VAR, SRC_CON, 22     ; add 22 (to get 64)
        divv   LOCAL_VAR, SRC_CON, 8      ; divide by 8 (to get 8)
                                        ; add itself to itself
        sumv   LOCAL_VAR, SRC_VAR, LOCAL_VAR
```

In addition to the commands shown above one can also subtract and multiply variables with a parameter, as well as getting the absolute (positive) value of a number and performing bit wise logical operations on the variables.

The example shows an important aspect of the Scout: It has both global and local (to a task) variables. This means that a task can use a set of variables for its own purposes without having to worry about the value being changed by another task and then use the global variables for information sharing with other tasks.

Outputs

Outputs (motors) have a polarity (direction) and a power level when turned on.

When turned off, the output can be floating or actively braking. Floating means that you remove the power supply to the output so that the robot/motor may continue on its own kinetic energy (inertia). Braking is when the output freezes in its current state meaning that motors stop instantly.

In the assembler commands, motors are addressed as a bit-list, i.e. output 1 has the value 0x01, output 2 has the value 0x02 and output 3 has the value 0x04. Combinations of motors (if you want a command to apply to more motors) are achieved by setting more bits in the list.

The program examples below show various settings of the motors:

```
#include “ScoutDef.h”

        pwr    OUTLIST_AB, SRC_CON, 7     ; power level ranges from 0 to 7
        dir    DIR_FWD, OUTLIST_A         ; forward here
        dir    DIR_RWD, OUTLIST_B         ; backwards here => spin round
        out    OUT_ON, OUTLIST_AB         ; go, go, go
```

The Scout also has some master control commands (assembler commands starting with ‘g’), which work similarly to these.

Speaker

A special output device is the integrated speaker, which can either play individual notes (given explicitly or read from a variable) or some of the built-in system sounds.

The resolution of notes is 10 ms (system sounds have fixed duration). Frequencies are given in hertz (Hz).

The program examples below show how to play various sounds:

```
#include “ScoutDef.h”

        playt  TONE_A5, FR_SEC_1

        plays  SND_ERROR

#define TONE_VAR     2

        setv   TONE_VAR, SRC_CON, TONE_A5
        mulv   TONE_VAR, SRC_CON, 2       ; ‘A’ raised an octave
        playv  TONE_VAR, FR_MS_500        ; Just half a second, please
```

The Scout does not buffer notes or system sounds, so it may be necessary to insert special wait commands to get the timing right if you want to recreate a tune.

Display

User programs cannot control the display on the Scout. In power mode (where the user can download programs) it will display a folder icon if there is a task 0 that can be started running.

While running, a series of icons will be animated as visual feedback.

During download of programs, another set of icons will be animated.

Inputs

The Scout has two external input ports to which touch sensors can be attached. The special color-coded ID touch sensors can also be used – they will automatically be detected by the Scout operating system. There is a third built-in input, which is a light sensor.

Inputs are typically read and stored in variables or used in comparisons for making decisions about program execution.

The program examples below show how to read and use sensor values:

```
#include "ScoutDef.h"


#define SENSOR_VAR   16

        setv   SENSOR_VAR, SRC_SENVAL, SEN_TOUCH1

        chk    SRC_CON, TVAL_PRESSED, EQ, SRC_VAR, SENSOR_VAR, ReleasedLabel

PressedLabel:
               {what to do when the touch sensor is pressed}
        jmp    ButtonFinishLabel

ReleasedLabel:
               {what to do when the touch sensor is not pressed}

ButtonFinishedLabel:
```

Instead of reading and storing the sensor value in a variable, one could ask the command to use the sensor value directly instead of a variable. This looks like:

```
        chk    SRC_CON, TVAL_PRESSED, EQ, SRC_SENVAL, SEN_TOUCH1, ReleasedLabel
```

For detailed information about where one can use what input sources and what values are within range for that input source, see the firmware and stand-alone mode specification document.

Events

The Scout firmware operating system has been extended to generate 'events' when important things happen. This can relieve the application from repeated testing for the situations directly and thus bring down both program size and system load.

Physical events

Physical events occur when the environment changes such as when a touch sensor (button) is pressed and released, when the light in the room is turned on and off, when a light is flashed into the light sensor a number of times, or when a message from another P-Brick is received.

Virtual events

Virtual events are events controlled by the downloaded programs. The program can set up timers and (score) counters. When a timer or counter reaches a predefined value, it generates an event (and the timer resets).

Handling events

It is always possible to see what events a task has received by looking at the TaskEventRegister (source type 23) and take appropriate actions.

The program can also wait for a specific event to happen before moving on.

Lastly it is possible to instruct the firmware operating system to start monitoring for specific events and then interrupt the task when they occur (after which the execution will continue at a predefined place in the program). See the "ENTER EVENT CHECK …" and "ENTER ACCESS CONTROL …" templates for specific information.

Structured design

Before writing any program, it is always useful to try to sketch out what the program is supposed to do, typically in the form of statements about normal behavior and what to do in case of external events.

Conditional behavior

If the behavior of the program is dependent on some external (a button being pressed or the light level being above a certain threshold) or internal (a variable having a certain value) condition, one should use the "IF … THEN … [ELSE …] ENDIF" template in the preceding chapter.

Repeated behavior

If the program needs to repeat a behavior a number of times or until a certain condition is fulfilled, one should use one of the LOOP, WHILE or UNTIL templates in the preceding chapter.

Interruptible behavior

Sometimes it is desirable to be able to react to external events immediately, without having to check for the situation continuously. To do this, the firmware operating system provides two different kind of 'monitors'. One monitor checks for events as described above, and the other monitor checks for access control to shared resources.

When an event happens (out of a program-selected set) or a resource is taken by another task with equal or higher priority, the normal program execution is stopped and restarted at a (different) program selected address, where proper action can be taken.

For these kinds of behavior, use the "ENTER EVENT CHECK … EXIT EVENT CHECK" and "ENTER ACCESS CONTROL … EXIT ACCESS CONTROL" templates in the preceding chapter.

Multi-tasking

The LEGO P-Bricks are multi-tasking which means that they can execute the tasks in the downloaded user programs in parallel. This enables the program to be broken up in independent chunks that each perform some piece of functionality and then interact with each other to deliver the overall performance.

The firmware operating system then executes commands in turn from the downloaded and active tasks. Each individual command is allowed to execute to completion before the operating system moves on to the next task or its own housekeeping activities. A task is active if it has been started and is not suspended by a wait command or is waiting for some event. The only task that is started when the green RUN button is pressed is task 0 – if more tasks are required, they must be started explicitly, either directly from task 0 or by a series of direct commands.

By breaking down a program in smaller pieces, each piece becomes easier to understand. The hard part is then to ensure that they interact correctly. Interaction is done by means of communication and synchronization.

Synchronization

For intra-task synchronization, one should use the monitors mentioned above – they also provide a form of inter-task synchronization.

For the most general form of inter-task synchronization, one has to use global variables. Since global variables are just that, global, care must be exercised when using them. The SEMAPHORE template shown in the preceding chapter provides a good safe access mechanism without the possibility of corrupting important data that other tasks may be using.

Task-to-task synchronization then consists in deciding a communication protocol using shared memory in the form of global variables.

A PC-to-P-Brick synchronization protocol is slightly more complex because the PC has to implement the SEMAPHORE scheme in its application as it cannot call subroutines directly. Since the PC is always Master (and the P-Brick is always Slave) in any communication one can implement a similar scheme using two global variables, one for flow control and one for data (the variables must be global as the PC cannot access variables that are local to a task).

Distributed systems

More complex systems can be constructed by programming several LEGO P-Bricks and having them exchange information by sending messages to each other. The same principles as above apply, only on a system-wide basis.

Communication

Messages are sent using the **msg** command.

When a message is received it generates an event. The actual value can be accessed using SOURCE 15.

When processed, the message buffer should be reset with the **msgz** command.

The message buffer is shared and global, so a little care should be exercised when using it.

Program Block Library (subroutines)

The Scout has an extensive subroutine library of general-purpose functions that can help reduce the size of downloaded user programs. Many of the subroutines expect parameters to be passed in local variables as outlined below.

The built-in subroutines are numbered 3-32 whereas user subroutines are numbered 0-2.

3 – MotorDriveSub (lvType)

Parameters:	lvType:	0: A Fwd, BFwd	5: A Rwd, B Off	9: C Fwd	(LocalVar1)
		1: A Rwd, B Rwd	6: A Off, B Fwd	10: C Rwd	
		2: A Fwd, B Rwd	7: A Off, B Rwd	11: C Off	
		3: A Rwd, BFwd	8: A Off, B Off		
		4: A Fwd, B Off			

Resources: lvType 0-8: Motor A, Motor B, lvType 9-11: VLL

Description: Set Motor AB or Motor C according to lvType. All local variables are preserved by the sub.

4 – BasicMotionSub (lvType, lvTime)

Parameters:	lvType:	1: Forward	2: ZigZag	(LocalVar1)
		3: CircleRight	4: CircleLeft	
		5: LoopA	6: LoopB	
		7: LoopAB		
	lvTime:	1-32767		(LocalVar2)

Resources: Motor A, Motor B

Description: Performs one loop of the basic motion types. lvTime sets the duration of each step in the motion. All local variables are preserved by the sub.

5 – AvoidSub (lvType, lvTime)

Parameters:	lvType:	0: AvoidLeft	(LocalVar1)
		1: AvoidRight	
	lvTime:	1-32767	(LocalVar2)

Resources: Motor A, Motor B, VLL

Description: Performs the avoid sequence avoiding right or left. Avoiding right has random turn time. lvTime, lvType and LocalVar4-8 are preserved by the sub.

6 – MovementsSub (lvType, lvTime)

Parameters:	lvType:	0: Dance	(LocalVar1)
		1: Bug	
		2: Random	
		3: Jitter	
	lvTime:	1-32767	(LocalVar2)

Resources: Motor A, Motor B, VLL, Sound

Description: Performs movement sequences and plays sounds.
lvTime and LocalVar4-8 are preserved by the sub.

7 – GetAverageLightSub ( )

Parameters: None

Resources: None

Description: Measures the light level averaged over 5 samples and returns it in lvAvrLight (LocalVar1)
LocalVar4-8 is preserved by the sub.

8 – AutoAdjustLightSub (lvCenterLight, lvThPercent, lvHPercent)

Parameters:			
	lvCenterLight	1-1020	(LocalVar1)
	lvThPercent	0-100	(LocalVar2)
	lvHPercent	0-100	(LocalVar3)

Resources: None

Description: Set LT, UT and H for the light sensor around lvCenterLight according to lvThPercent and lvHPercent.
lvCenterLight, lvThPercent, lvHPercent and LocalVar5-8 are preserved by the sub.

9 – SeekSub (lvType, lvTime)

Parameters:			
	lvType:	0: SeekDark 1: SeekLight	(LocalVar1)
	lvTime:	1-32767	(LocalVar2)

Resources: Motor A, Motor B, Sound

Description: Finds the direction of lowest or highest light intensity.
lvTime and LocalVar6-8 are preserved by the sub.

10 – FindBrightSub (lvBrightTH, lvBrightSteps)

Parameters:			
	lvBrightTH:	1-1020	(LocalVar1)
	lvBrightSteps:	1-32767	(LocalVar2)

Resources: Motor A, Motor B, Sound

Description: Finds the direction with a light level lower than lvBrightTH. Samples lvBrightSteps times.
lvBrightTH and LocalVar5-8 are preserved by the sub.

11 – GetMotorStatusSub ( )

Parameters: None

Resources: None

Description: Gets the immediate state of motors A and B. StatusA returned in LocalVar1, StatusB in LocalVar2. Status: 0: Off, 1: Fwd, 2: Rwd.
LocalVar3-8 are preserved by the sub.

12 – Motor2SoundSub (lvStatusA, lvStatusB)

Parameters: lvStatusA: 0-2 (LocalVar1)
lvStatusB: 0-2 (LocalVar2)

Resources: Sound

Description: Plays a system sound according to lvStatusA and lvStatusB.
All local variables are preserved by the sub.

13 – LightGeigerSub (lvIntgLimit)

Parameters: lvIntgLimit: 1-32767 (LocalVar1)

Resources: Sound

Description: Enters looping forever playing a sequence of beeps at a rate proportional to the light level.
In the loop (kLightOffset – LightValue) will be added to the integrator followed by a 10ms Wait.
If the integrator exceeds lvIntgLimit a tone of kGeigerL2F * (kToneOffset – LightValue) Hz is played for 10ms and the integrator is reset.
lvIntgLimit and LocalVar4-8 are preserved by the sub.

14 – FwdSub (lvDuration, lvTaskFlags)

Parameters: lvDuration: -32768 -32767 (LocalVar1)
lvTaskFlags: Bit15: 0-1 (LocalVar8)

Resources: Motor A, Motor B

Description: If Bit15 in lvTaskFlags is set Access Control is set up.
Sets up motor control: MotorA Fwd, MotorB Fwd.
After motor control is set up, the duration of the Sub is determined:
lvDuration
- > 0 A Wait of lvDuration*10ms is performed
- < 0 lvDuration is treated as an event list and a WaitUntilEvent is performed
- = 0 Enters Looping forever

lvDuration and LocalVar3-8 are preserved by the sub.

The following subs works in the same way as FwdSub, but set up different motor control:

15 – RwdSub (lvDuration, lvTaskFlags)

MotorA Rwd, MotorB Rwd

16 – SpinRightSub (lvDuration, lvTaskFlags)

MotorA Fwd, MotorB Rwd

17 – SpinLeftSub (lvDuration, lvTaskFlags)

MotorA Rwd, MotorB Fwd

18 – FwdTurnRightSub (lvDuration, lvTaskFlags)

MotorA Fwd, MotorB Off

19 – RwdTurnLeftSub (lvDuration, lvTaskFlags)

MotorA Rwd, MotorB Off

20 – FwdTurnLeftSub (lvDuration, lvTaskFlags)

MotorA Off, MotorB Fwd

21 – RwdTurnRightSub (lvDuration, lvTaskFlags)

MotorA Off, MotorB Rwd

22 – ZigZagSub (lvDuration, lvTime, lvTaskFlags)

Parameters:	lvDuration:	-32768 -32767	(LocalVar1)
	lvTime:	1-32767	(LocalVar2)
	lvTaskFlags:	Bit15: 0-1	(LocalVar8)

Resources: Motor A, Motor B

Description: If Bit15 in lvTaskFlags is set Access Control is set up.
Does the ZigZag motion with lvTime between the steps.
lvDuration
- > 0 ZigZag lvDuration times (loop)
- < 0 lvDuration is treated as an event list and the ZigZag is performed until the event happens
  After expired duration Motor A and B are floated
- = 0 ZigZags forever

lvDuration, lvTime and LocalVar5-8 are preserved by the sub.

23 – CircleRightSub (lvDuration, lvTime, lvTaskFlags)

Parameters:	lvDuration:	-32768 -32767	(LocalVar1)
	lvTime:	1-32767	(LocalVar2)
	lvTaskFlags:	Bit15: 0-1	(LocalVar8)

Resources: Motor A, Motor B

Description: If Bit15 in lvTaskFlags is set Access Control is set up.
Does the CircleRight motion with lvTime between the steps.
lvDuration
- > 0 Repeats CircleRight step lvDuration times (loop)
- < 0 lvDuration is treated as an event list and the CircleRight step is performed until the event happens
  After expired duration Motor A and B are floated
- = 0 Repeats CircleRight steps forever

lvDuration, lvTime and LocalVar5-8 are preserved by the sub.

24 – CircleLeftSub (lvDuration, lvTime, lvTaskFlags)

Parameters:	lvDuration:	-32768 -32767	(LocalVar1)
	lvTime:	1-32767	(LocalVar2)
	lvTaskFlags:	Bit15: 0-1	(LocalVar8)

Resources: Motor A, Motor B

Description: If Bit15 in lvTaskFlags is set Access Control is set up.
Does the CircleLeft motion with lvTime between the steps.
lvDuration
- > 0 Repeats CircleLeft step lvDuration times (loop)
- < 0 lvDuration is treated as an event list and the CircleLeft step is performed until the event happens
  After expired duration Motor A and B are floated
- = 0 Repeats CircleLeft steps forever

lvDuration, lvTime and LocalVar5-8 are preserved by the sub.

25 – AvoidRightSub (lvMovTime, lvTaskFlags)

Parameters: lvMovTime: 1 -32767 (LocalVar1)
lvTaskFlags: Bit15: 0-1 (LocalVar8)

Resources: Motor A, Motor B

Description: If Bit15 in lvTaskFlags is set Access Control is set up.
Does the AvoidRight motion with lvMovTime between the steps.
All local variables except LocalVar2 are preserved by the sub.

26 – AvoidLeftSub (lvMovTime, lvTaskFlags)

Parameters: lvMovTime: 1 -32767 (LocalVar1)
lvTaskFlags: Bit15: 0-1 (LocalVar8)

Resources: Motor A, Motor B

Description: If Bit15 in lvTaskFlags is set Access Control is set up.
Does the AvoidLeft motion with lvMovTime between the steps.
All local variables except LocalVar2 are preserved by the sub.

27 – BugshakeSub (lvMovTime, lvTaskFlags)

Parameters: lvMovTime: 1 -32767 (LocalVar1)
lvTaskFlags: Bit15: 0-1 (LocalVar8)

Resources: Motor A, Motor B

Description: If Bit15 in lvTaskFlags is set Access Control is set up.
Does the Bugshake motion with lvMovTime between the steps.
All local variables except LocalVar2 are preserved by the sub.

28 – LoopABSub (lvMovTime, lvTaskFlags)

Parameters: lvMovTime: 1 -32767 (LocalVar1)
lvTaskFlags: Bit15: 0-1 (LocalVar8)

Resources: Motor A, Motor B

Description: If Bit15 in lvTaskFlags is set Access Control is set up.
Does the LoopAB motion with lvMovTime between the steps.
All local variables except LocalVar2 are preserved by the sub.

29 – GetSema0Sub ( )

Parameters: None

Resources: None

Description: Try to get access to a resource through gvSema0.
All local variables are preserved by the sub.

The following subs works in the same way as GetSema0Sub, but uses different global variables:

30 – GetSema1Sub ( )

Uses GlobalVar1.

31 – GetSema1Sub ( )

Uses GlobalVar1.

32 – InitSysSub ( )

Parameters: None

Resources: None

Description: Initializes system registers.
All local variables are preserved by the sub.

VLL Command Set

The VLL command set consists of 128 distinct byte codes. Each code is interpreted in the receiving device and can have different meanings in each device. A convention has been made though to have separate code groups for motor commands and sound commands. For the MicroScout, ‘D’ signifies Direct commands and ‘S’ signifies Scripting commands.

VLL code	Code Pilot	MicroScout
0	Motor Forward	D: Motor Forward
1	Motor Reverse	D: Motor Reverse
2		
3		
4	Sound (Valve)	D: Beep 1
5	Sound (Helicopter)	D: Beep 2
6	Sound (Truck)	D: Beep 3
7	Sound (Robot)	D: Beep 4
8	Sound (Machine)	D: Beep 5
9	Sound Mute	
10	Motor Stop	D: Motor Stop
11	Motor & Sound	
12	Fixed Prgm Truck	
13	Fixed Prgm Wheel Driver	
14	Fixed Prgm Crash Buggy	
15	Fixed Prgm Robot	
16		S: Motor Forward 0.5
17		S: Motor Forward 1.0
18		S: Motor Forward 2.0
19		S: Motor Forward 5.0
20		S: Motor Reverse 0.5
21		S: Motor Reverse 1.0
22		S: Motor Reverse 2.0
23		S: Motor Reverse 5.0
24		S: Beep 1
25		S: Beep 2
26		S: Beep 3
27		S: Beep 4
28		S: Beep 5
29		S: Wait for Light
30		S: Seek Light
31		S: Code
32		S: Keep Alive
33		D: Run
34		D: Delete Script
35		
36		
37		
38		
39		
40		
41		
42		
43		
44		
45		
46		

VLL code	Code Pilot	MicroScout
47		
48		
49		
50		
51		
52		
53		
54		
55		
56		
57		
58		
59		
60		
61		
62		
63		
64		
65		
66		
67		
68		
69		
70		D: Next
71		D: Reset
72		
73		
74		
75		
76		
77		
78		
79		
80		
81		
82		
83		
84		
85		
86		
87		
88		
89		
90		
91		
92		
93		
94		
95		
96	Touch In	
97	Touch Out	
98		
99	Tone C	
100	Tone C#	

VLL code	Code Pilot	MicroScout
101	Tone D	
102	Tone D#	
103	Tone E	
104	Tone F	
105	Tone F#	
106	Tone G	
107	Tone G#	
108	Tone A	
109	Tone A#	
110	Tone H (B)	
111	Tone C	
112	Number 0	
113	Number 1	
114	Number 2	
115	Number 3	
116	Number 4	
117	Number 5	
118	Number 6	
119	Number 7	
120	Number 8	
121	Number 9	
122	Decimal dot	
123	Random	
124	Speed/Torque Low (20)	
125	Speed/Torque Med (40)	
126	Speed/Torque High (60)	
127	Tacho	